

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER: H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees, H. Niemann,
D. Seitzer, B. Schmidt, H. J. Schneider, H. Wedekind

FORMALE BESCHREIBUNG DES INKREMENTELLEN PROGRAMMIERENS-IM-GROSSEN MIT GRAPH-GRAMMATIKEN

Rupert Gall

DISSERTATION

BAND 16 · NUMMER 1 · ERLANGEN · FEBRUAR 1983

ISSN 0344 - 3515



Digitized by the Internet Archive
in 2026

FORMALE BESCHREIBUNG DES INKREMENTELLEN
PROGRAMMIERENS-IM-GROSSEN
MIT GRAPH-GRAMMATIKEN

Der Technischen Fakultät
der Universität Erlangen-Nürnberg
zur Erlangung des Grades
DOKTOR - INGENIEUR

vorgelegt
von

Rupert Gall

Erlangen 1982



Als Dissertation genehmigt von der
Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung: 14. 12. 1982

Tag der Promotion: 18. 02. 1983

Dekan: Prof. Dr. H.-U. Zwicker

Berichterstatter: Prof. Dr. H. J. Schneider
Prof. Dr. H. Müller

ZUSAMMENFASSUNG

Der Software-Arbeitsplatz IPSEN (Incremental Programming Support Environment) unterstützt die Entwicklung "wohlstrukturierter" Softwaresysteme vom Systementwurf bis zum Systemtest. Die zu IPSEN integrierten Werkzeuge bieten eine einheitliche Benutzerschnittstelle und basieren auf graphförmigen internen Datenstrukturen.

Die vorliegende Arbeit beschreibt die IPSEN-Werkzeuge fürs Programmieren-im-Großen.

Ein "wohlstrukturiertes" Softwaresystem setzt sich aus hierarchisch angeordneten funktions- und datenorientierten Systemkomponenten zusammen. Der Informationsfluß zwischen Systemkomponenten wird durch Vergabe von Zugriffs- bzw. Benutzungsrechten kontrolliert. Die Struktur eines zu entwickelnden Softwaresystems hält IPSEN im sog. Systemgraph fest.

Die in einer speziellen Spezifikationssprache formulierte Spezifikation der syntaktischen Systemstruktur kann batchorientiert oder inkrementell eingebracht und geprüft werden. Die korrespondierenden Systemgraph-Transformationen werden durch eine programmierte zweistufige attributierte Graph-Grammatik beschrieben. Die anschließende automatische Umsetzung der Systemspezifikation in eine Spezifikation von Programm-Moduln der Implementierungssprache wird an den Beispielen ADA und FORTRAN 77 aufgezeigt.

Für den Systemtest nach der Implementierung der Programm-Moduln bietet IPSEN neben der konventionellen Systemausführung (Übersetzen, Binden, Laden) auch eine interpretative Systemausführung (dynamisches Binden und Laden) durch Interpretation des Systemgraphen.

IPSEN bestimmt exakt die Auswirkung nachträglicher inkrementeller Modifikationen der Systemspezifikation und kontrolliert die Wiederherstellung der Systemkonsistenz.

Am Ende der Arbeit wird noch kurz auf phasenübergreifende IPSEN-Werkzeuge eingegangen.

1	Einführung	1
1.1	Einleitung	1
1.2	Phasen der Systementwicklung	3
1.3	Ein Software-Arbeitsplatz	6
2	Wohlstrukturierte Softwaresysteme	11
2.1	Prinzipien des Systementwurfs	11
2.1.1	Modularisierung	12
2.1.2	Hierarchiebildung	15
2.1.3	Kontrollierte Informationsverteilung	17
2.2	Strukturmodell	18
2.2.1	ASSMEM - ein Software-Assoziativspeicher	19
2.2.2	Systemkomponenten	20
2.2.2.1	Abstraktes Datenobjekt	21
2.2.2.2	(Generischer) abstrakter Datentyp	26
2.2.2.3	Typkonzept	36
2.2.2.4	Dynamische Datenstrukturen	38
2.2.2.5	Abstrakte Funktion	39
2.2.2.6	(Parametrisierte) Funktionsklasse	41
2.2.3	Beziehungen zwischen Systemkomponenten	47
2.2.3.1	Systemhierarchie	47
2.2.3.2	Zugriffsrecht-Beziehung	51
2.2.3.3	Benutzungsrecht-Beziehung	63
2.2.4	Detailaspekte	68
2.2.4.1	Parameterarten bei generischen abstrakten Datentypen	68
2.2.4.2	Benutzungsspezifikation	74
2.2.4.3	Deklarationen datenorientierter Systemkomponenten	75
2.2.4.4	Aktuelle Benutzung	75
2.2.4.5	Namen von Systemkomponenten	77
2.2.5	Realisierungs-Varianten	77

	Seite
3 Formale Beschreibung des Systemgraphen	81
3.1 Graphentheoretische Grundlagen	81
3.2 Kontextfreie Syntax eines Systemgraphen	82
3.3 Konsistenzbedingungen	83
3.3.1 Korrektheit der Realisiert-Hierarchie . . .	84
3.3.2 Korrektheit der expliziten Zugriffsrecht-Be- ziehung	85
3.3.3 Korrektheit der expliziten Benutzungsrecht- Beziehung	91
3.3.4 Zugriffsrecht-Konsistenz	94
3.3.5 Weitere Konsistenzbedingungen	96
4 Erstellung der Systemspezifikation	97
4.1 Rechnerunterstützung bei der Erstellung der Spezi- fikation	97
4.2 Batchorientierte Erstellung	97
4.2.1 Getrennte Spezifizierbarkeit	99
4.2.2 Entwurfsstrategien	108
4.2.3 Anmerkungen zum Unparse-Verfahren	113
4.3 Inkrementelle und syntaxgesteuerte Erstellung . . .	114
4.3.1 Benutzerschnittstelle	116
4.3.1.1 Begriff des Inkrements	116
4.3.1.2 Inkrementoperationen	118
4.3.1.3 Syntaxgesteuerter Spezifikations- Editor	119
4.3.2 Implementierungskonzept	126
4.3.2.1 Graph-Grammatik für Systemgraph- Transformationen	127
4.3.2.2 Implementation des Spezifikations- Editors	131
4.3.2.3 Erweiterung des Systemgraphen . . .	135
4.3.3 Beispiel für die Eingabe einer Spezifika- tionseinheit	137
5 Von der Systemspezifikation zum ausführbaren System . .	152
5.1 Codieren-im-Großen	152
5.1.1 Übertragung nach ADA	154
5.1.2 Übertragung nach FORTRAN 77	162

	Seite
5.2 Implementieren eines Programm-Moduls	167
5.3 Systemausführung	173
5.3.1 Ausführungsmodi	173
5.3.2 Ausführungsüberwachung	177
6 Nachträgliche inkrementelle Modifikation der Systemspezifikation	179
6.1 Durchführung einer Modifikation	179
6.2 Inkrementeingabe in einer SK-Spezifikation	181
6.3 Inkrementänderung und -löschung in einer SK-Spezifikation	191
6.4 Inkrementeingabe in einer Realisierungs-Variante	200
6.5 Inkrementänderung und -löschung in einer SK-Realisierungs-Variante	204
6.6 Wiederherstellung der Ausführbarkeit	205
7 Phasenübergreifende Werkzeuge und Literaturvergleich	208
Schlußbemerkung	211
Stichwortverzeichnis	212
Literaturverzeichnis	215
Anhang: Grammatik der Spezifikationssprache	

KAPITEL 1: EINFÜHRUNG

1.1 EINLEITUNG

Die Komplexität heutiger Softwaresysteme streift häufig die Grenze des menschlich Überschaubaren. Mit dem Umfang eines Softwaresystems wächst meist der Vernetzungsgrad der einzelnen Systemkomponenten und Teilsysteme. Es besteht die Gefahr eines nur schwer kontrollierbaren Daten- und Kontrollflusses.

Mangelhafte Strukturierung der zu entwickelnden Softwaresysteme war Hauptursache für die bekannte "Software-Krise", die sich Ende der sechziger Jahre offenbarte, als Softwaresysteme komplexer wurden. Nur schwach ausgeprägte Modularisierung erhöht den Aufwand und damit die Kosten für die Systementwicklung und -wartung.

Man erkannte als Ausweg aus der Krise die Notwendigkeit, die Entwicklung komplexer Software soweit wie möglich ingenieurmäßig zu betreiben. Dies bedeutet zum einen, für den Entwurf eines Softwaresystems Methoden und Prinzipien vorzugeben, die Struktur in das System einbringen. Zum anderen sind rechnergestützte Werkzeuge bereitzustellen, die es dem Systementwickler gestatten, sich auf die intellektuell anspruchsvollen Tätigkeiten zu konzentrieren, um somit seine Produktivität insgesamt zu erhöhen.

Konventionelle Programmiersysteme, bestehend aus Texteditor, Compiler, Binder, Lader und Dateisystem, bieten noch kein Optimum an Unterstützung. Diese Werkzeuge unterstützen nur die Systemimplementierung, nicht jedoch den Systementwurf und die Entwicklungsphasen nach der Implementierung. Sie kommunizieren i.a. über unterschiedlich angelegte Intern-Schnittstellen und bieten dem Entwickler keine einheitliche Benutzerschnittstelle. Die genannten Werkzeuge sind außerdem nur auf einen einzelnen Entwickler zugeschnitten, berücksichtigen also nicht, daß Softwareentwicklung heute einer Projektorganisation wie bei anderen ingenieurmäßig durchzuführenden Projekten bedarf.

Ein moderner Software-Arbeitsplatz vermeidet die angedeuteten Nachteile eines konventionellen Programmiersystems, indem er die Entwicklung strukturierter modularer Softwaresysteme (vom Entwurf bis zur Wartung) durch ein ganzes Entwicklerteam unterstützt und seine Leistungen in einer einheitlichen projektorien-

tierten Sichtweise anbietet. Eine solche Software-Entwicklungs-umgebung wird in dem an der Universität Osnabrück laufenden Forschungsprojekt IPSEN (Incremental Programming Support Environment) in Zusammenarbeit mit der Universität Erlangen-Nürnberg konzipiert und an der Universität Osnabrück implementiert.

Die vorliegende Arbeit enthält das Konzept für die Entwicklungs-Werkzeuge, die von IPSEN für den Entwurf modularer Softwaresysteme, ihren Test und ihre Wartung bereitgehalten werden. Für die formale Beschreibung dieser Entwicklungs-Werkzeuge dient eine spezielle Graph-Grammatik.

Diese Arbeit ist auch im Rahmen der Forschungstätigkeiten an der Universität Erlangen-Nürnberg zu sehen, die sich die Spezifikation inkrementeller Compiler durch Graph-Grammatiken zum Ziel setzten (/SCH75b/,/NAG80/).

In den weiteren Abschnitten des ersten Kapitels werden die bei einer Systementwicklung durchlaufenen Entwicklungsphasen präzisiert und die Funktionen detailliert aufgeführt, die ein Software-Arbeitsplatz dafür anbieten sollte. Im zweiten Kapitel werden die Systemkomponenten mit ihren Querbeziehungen diskutiert, die das zu entwickelnde Softwaresystem zu einem "wohlstrukturierten" System werden lassen. Der Systemgraph als interne Datenstruktur unserer Entwicklungs-Werkzeuge wird formal im dritten Kapitel beschrieben. Im vierten Kapitel wird die rechnergestützte Erstellung der Systemspezifikation als Entwurfsergebnis, im fünften Kapitel der Übergang von der Systemspezifikation zur Implementierung und zur Ausführung des Systems besprochen. Welche Auswirkungen nachträgliche Modifikationen an der Systemspezifikation haben, wird im sechsten Kapitel erörtert. Das siebte Kapitel schließlich enthält eine Auflistung der IPSEN-Werkzeuge für Aufgaben, die sich aus der Projektorganisation ergeben, und einen Vergleich mit anderen Software-Entwicklungsumgebungen.

1.2 PHASEN DER SYSTEMENTWICKLUNG

Der Übergang von einem gegebenen Problem zu einem ablauffähigen Softwaresystem als entsprechende Problemlösung kann grob in folgenden Tätigkeiten und ihnen zugeordnete Ergebnisdokumente unterteilt werden.

Die erste Tätigkeit, die *Problem-analyse*, umfaßt die geistige Durchdringung und Erfassung des Problems und resultiert in einer mehr oder weniger formalen *Problembeschreibung* (*Anforderungsdefinition*).

Basierend auf dieser Problembeschreibung wird das System *entworfen*, d.h. die Systemstruktur festgelegt. Im Entwurf beschäftigt sich der Entwickler zum einen mit rein syntaktischen Aspekten: Fragen wie "wieviele Systemkomponenten welcher Art sind einzuführen", "welche Systemkomponenten bilden Teilsysteme innerhalb des Gesamtsystems" oder "welche Systemkomponenten/Teilsysteme interagieren in welcher Form (Benutzung von Ressourcen, Austausch von Daten usw.) miteinander" beziehen sich ausschließlich auf die Fixierung der *Syntax der Systemstruktur*.

Zum anderen wird die Semantik (Funktion) der Systemkomponenten/Teilsysteme und ihrer Vernetzung spezifiziert (*Semantik der Systemstruktur*). Es wird beispielsweise (evtl. formal) festgehalten, daß eine bestimmte Systemkomponente

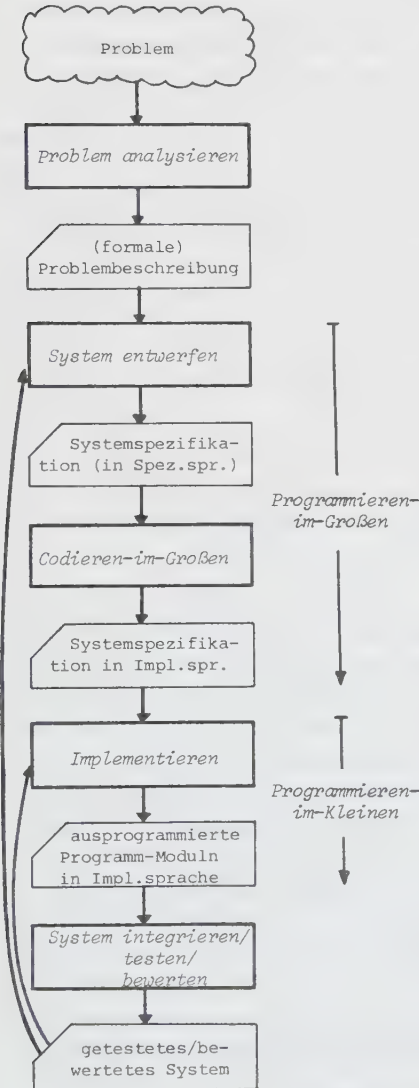


Abb. 1-1

einen Ringpuffer mit seinen typischen Bearbeitungsoperationen (und nicht etwa einen Keller) zu realisieren hat bzw. bei Nebenläufigkeit der Austausch bestimmter Informationen zwischen zwei Systemkomponenten der Synchronisierung dient.

Der Entwurfsprozeß läuft i.a. in mehreren Iterationszyklen ab. Sein Ergebnis wird in der *syntaktischen* und *semantischen Systemspezifikation* festgehalten. Ist diese in einer Spezifikationssprache formal formuliert, kann die (syntaktische und semantische) Systemspezifikation bereits zur Entwurfszeit auf Einhaltung gewisser Syntax- und Konsistenzbedingungen überprüft werden. Die *syntaktische Systemspezifikation* ist als Vorgabe für die Implementierung einzelner Systemkomponenten/Teilsysteme und ihrer Integration zum Gesamtsystem relevant. Die *semantische Systemspezifikation* dagegen dient (außer als Implementierungs-Vorgabe) nach der Implementierung bzw. Integration der Systemkomponenten als Vorgabe für den Nachweis semantischer Korrektheit (z.B. mittels Test oder Verifikation). Für die vorliegende Arbeit ist nur die syntaktische Spezifikation eines Systems (syntaktische Systemstruktur)¹⁾ bedeutsam, auf die semantische Spezifikation wird im folgenden nur mehr am Rande eingegangen.

Bei der Entwicklung umfangreicher Softwaresysteme wird die Systemspezifikation von einem i.a. hierarchisch organisierten Entwicklungsteam erstellt.

Die Spezifikation der Systemstruktur wird im weiteren System-Erstellungsprozeß in eine Form übertragen, die sich ausschließlich der Sprachkonstrukte der gewählten Implementierungssprache bedient (z.B. Modulkonzept in MODULA-2 /WIR80/, Package-Konzept in ADA /DoD80/). Diesen Übertragungsvorgang nennen wir *Codieren-im-Großen*. Der Aufwand für diese Übertragung und die Transparenz der resultierenden implementierungssprachlichen Spezifikation hängt unmittelbar davon ab, inwieweit das Konzept der Implementierungssprache u.a. Unterstützung für die Erstellung großer Programmsysteme bietet. Liegt die Ausgangs-Spezifikation formal formuliert vor, läßt sich die Übertragung weitgehend automatisieren.

Die Tätigkeiten des Entwerfens bzw. späteren Änderns²⁾ der System-

1) Im weiteren wird der Begriff *Systemspezifikation* nur mehr in diesem Bedeutungs-Zusammenhang verwendet; ebenso stehe *Systemstruktur* stets ausschließlich für die syntaktische Systemstruktur.

2) Zur Fehlerbeseitigung bzw. im Zuge der Systempflege.

struktur und des Codierens-im-Großen fassen wir als *Programmieren-im-Großen* zusammen.¹⁾ Das Ergebnis, die in der Implementierungssprache formulierte Systemspezifikation, besteht in erster Linie aus den Rahmen von Programm-Moduln, die später noch auszuprogrammieren sind. In diesen Moduln sind bereits die zu ex-/importierenden Ressourcen und lokale Größen vereinbart, die zur Realisierung der Export-Ressourcen benötigt werden. Die Semantik einer Systemkomponente und ihrer Interaktion mit dem Restsystem kann im entsprechenden Programm-Modul in Kommentar-Form als Implementierungs-Vorgabe eingebracht werden.

Im Zuge des *Programmierens-im-Kleinen*, das sich an das *Programmieren-im-Großen* anschließt, werden die in der Implementierungssprache spezifizierten Programm-Moduln ausprogrammiert. Anschließend liegt das zu entwickelnde Softwaresystem in *Quellfassung* vor (*Systemimplementation*).

Die Implementierung der Programm-Moduln wird bei größeren Softwaresystemen i.a. von mehreren Mitgliedern des Entwicklungsteams durchgeführt.

Nach der Übersetzung und dem erfolgreichen Test einzelner Programm-Moduln/Teilsysteme²⁾ werden in einem Bindevorgang die *Bindecodes* der Programm-Moduln zum Gesamtsystem integriert (*Systemintegration*). Sodann wird das Gesamtsystem gemäß einer vorgegebenen Teststrategie auf seine funktionale Korrektheit hin überprüft (*Systemtest*), wobei die Anforderungsdefinition Testkriterium ist.

Fehlerkorrekturen können Rückgriffe auf die Phase "Implementieren" (modul-lokale Fehler), aber auch auf den Systementwurf (Modifikation von Schnittstellen zwischen Moduln, nachträglicher Einbau unberücksichtigt gebliebener Systemfunktionen u.ä.) nötig machen.

Die *Systempflege* dient der Leistungsverbesserung des Systems aufgrund einer durchgeführten Leistungsbewertung (z.B. Steigerung der Laufzeit-Effizienz), aber auch der Anpassung des Systems an neue Benutzerwünsche, also an eine neue Anforderungsdefinition (z.B. abgemaßerte System-Variante, kürzere Reaktionszeiten). Beide Aspekte können durch Modifikationen auf der Modul-Ebene abgefangen werden (z.B. Austausch eines Moduls durch einen effizienter implementierten), aber auch eine Um-/Neustrukturierung eines Teilsystems oder des Ge-

¹⁾ Der in /DEKR76/ erstmals geprägte Begriff "Programmieren-im-Großen" umfaßt dort nur den ersten Aspekt unserer Sichtweise.

²⁾ Testkriterium ist die semantische Systemspezifikation.

samtsystems (z.B. Optimierung des systeminternen Daten- und Kontrollflusses) erfordern, was meist zeit- und deshalb kostenintensiv ist.

Die aufgezeigten Rückgriffe auf frühere Aktivitäten und deren Ergebnisse rechtfertigen den Begriff *Software-Entwicklungszyklus*.

Im folgenden Abschnitt wird angedeutet, inwieweit die zum Programmieren-im-Großen und den anschließenden Entwicklungsphasen zählenden Aktivitäten durch Rechnereinsatz unterstützt werden können, um die Produktivität des Entwicklungsteams und die Qualität des Endprodukts zu steigern.

1.3 EIN SOFTWARE-ARBEITSPLATZ

Rechnerunterstützung im Zuge der Software-Entwicklung ist hauptsächlich durch zwei Gesichtspunkte motiviert.

Zum einen bietet der Rechnereinsatz verstärkt die Möglichkeit, Softwaresysteme zu entwickeln, die bestimmten Strukturierungsregeln genügen. Systeme, die gemäß einem standardisierend wirkenden Strukturmodell entworfen werden, besitzen erhöhte Struktur-Transparenz und sind deshalb i.a. zuverlässiger und für Änderungen leichter zugänglich.

Zum anderen kann der Systementwickler (bzw. das Systementwicklungsteam) während des Programmierens-im-Großen und den sich anschließenden Entwicklungsphasen in vielen, vor allem verwaltungsbedingten Aktivitäten entlastet werden, so daß er seine Kreativität verstärkt anspruchsvolleren Aufgaben zuwenden kann. Das zu entwickelnde System hat in diesem Entwicklungsstadium bereits einen Grad an Formalheit erreicht, der umfassende Rechnerunterstützung ermöglicht, wodurch die Effizienz der Systemerstellung gesteigert wird.

Unterstützen einheitlich konzipierte und aufeinander abgestimmte Werkzeuge das Programmieren-im-Großen und das Programmieren-im-Kleinen, so realisieren sie in ihrer Gesamtheit einen *Software-Arbeitsplatz* (programmer's workbench). Synonyme dazu sind *Software-Entwicklungsumgebung* (software development environment, software engineering environment) oder *Software-Erstellungswerkzeuge*.

Programm-Entwicklungsumgebungen (program development environment)

oder *Programm-Erstellungswerkzeuge* hingegen unterstützen mehr die Entwicklung einzelner "autonomer" Programme im klassischen Sinne, sind also ausschließlich dem Programmieren-im-Kleinen zuzuordnen.

Ein moderner *Software-Arbeitsplatz* sollte den technischen Teil der Software-Entwicklung zumindest insofern unterstützen, als er die Entwicklung ausschließlich "wohlstrukturierter" Softwaresysteme fördert, Eingabe und Modifikation von Systemspezifikation und -implementation syntaxgesteuert und inkrementell durchzuführen gestattet, die Systemkonsistenz überwacht, komfortable Testhilfen bietet und die Varianten-/Versionskontrolle übernimmt.

Sämtliche Funktionen des Software-Arbeitsplatzes sind in einer einheitlich gestalteten Benutzerschnittstelle anzubieten. Ein-/Ausgabemedium ist stets der Bildschirm.

Ist der Software-Arbeitsplatz auf einem eigenen Entwicklungsrechner beispielsweise auf der Basis von Minicomputer-Hardware realisiert, ist eine zielrechnerunabhängige Software-Entwicklung möglich. Erst nach erfolgreichem Test wird die entwickelte Software in Quell- oder Bindecode-Form auf den Zielrechner übertragen.

Das Konzept des Software-Arbeitsplatzes IPSEN erfüllt die an einen modernen Software-Arbeitsplatz gestellten Forderungen. Im folgenden werden die geplanten Leistungen stichpunktartig charakterisiert:

1. Phasenspezifische Unterstützung

Programmieren-im-Großen

Systementwurf

- erzwingt die Konstruktion von ausschließlich "wohlstrukturierten" Systemen durch
 - syntaxgesteuertes/inkrementelles oder batchorientiertes Entwerfen¹⁾ der einzelnen Systemkomponenten
 - Überwachung der Strukturkonsistenz hinsichtlich bestimmter Systemstrukturierungs-Prinzipien bei beliebiger Entwurfsstrategie (top-down, bottom-up, gemischt)

¹⁾ "Entwerfen" umfasse (wie bisher) nur syntaktische, keine semantischen Aspekte.

- Vollständigkeitsprüfungen
- stellt Funktionen zur Strukturanalyse bereit.

Codieren-im-Großen

- ermöglicht Vorgaben für die Implementierung der Programm-Moduln (Modul-Rahmen, benutzbare Ressourcen, Prozedur-köpfe, ...).

Programmieren-im-Kleinen

- sieht die syntaxgesteuerte, inkrementelle Erstellung und Änderung einzelner (modul-lokaler) MODULA-2-Programme vor (syntaxgesteuerter Editor, inkrementeller Compiler); überprüft für jedes Inkrement sofort die Konsistenz mit der entsprechenden Modul-Spezifikation.
- überprüft im batchorientierten Erstellungsmodus ausprogrammierte Programm-Moduln gegen die Modul-Spezifikation (konventionelles Programmieren-im-Kleinen).
- stellt ein komfortables sprachbezogenes Validierungs- und Debug-System bereit.

Systemintegration/-test/-pflege

- unterstützt die konventionelle Systemausführung (Übersetzen, Binden, Laden) bzw. die partielle Ausführung von Teilsystemen eines noch unvollständigen Gesamtsystems.
- ermöglicht sprachbezogenes Testen und Debugging auf Modul- bzw. Teilsystem-Ebene.
- bietet Funktionen zur Analyse des implementierten Systems und seines Laufzeitverhaltens an.
- bestimmt die Teilsysteme, die von der Modifikation einer Modul-Schnittstelle betroffen sind.
- stößt nach Modifikationen Maßnahmen zur Wiederherstellung der System-Konsistenz an (Meldungen an Entwickler betroffener Moduln, evtl. automatische Neuübersetzung).

2. Phasenübergreifende Unterstützung

Varianten-/Versionskontrolle

- verwaltet die verschiedenen Varianten eines Programm-Moduls (Dummy-Variante, Varianten aufgrund verschiedener zugrundeliegender Algorithmen, laufzeit- bzw. speicherplatz-optimale Variante u.a.) bzw. deren Versionen (Quellversion, Bindeversion, private bzw. freigegebene Version u.a.).
- übernimmt die Zugriffs- und Freigabekontrolle.
- generiert automatisch verschiedene System-Varianten bei Vorgabe bestimmter System-Attribute.

Dokumentation

- bietet Hilfen für die Erstellung der Ergebnisdokumente zu sämtlichen Entwicklungsphasen.

Projektmanagement

- unterstützt die Projektplanung, die Abwicklung administrativer Aufgaben, die Terminkontrolle, die Erstellung der Projektstatistik etc.

Projektorganisation

- unterstützt die Kommunikation zwischen den Entwicklern eines Teams.
- verwaltet die Projektbibliothek.

In der vorliegenden Arbeit werden die Werkzeuge von IPSEN konzipiert, die das Programmieren-im-Großen, Systemintegration/-test/-pflege und die Variantenkontrolle unterstützen. Dabei wird jedoch nur *sequentielle* Programmierung-im-Großen betrachtet, also keine zusätzliche Unterstützung für Softwaresysteme vorgesehen, die auf Nebenläufigkeit basieren.

Da - wie sich noch zeigen wird - Programmieren-im-Großen und -im-Kleinen oftmals nicht exakt gegeneinander abgrenzbar sind, wird an einigen Stellen auch auf die Werkzeuge für das Programmieren-im-Kleinen eingegangen.

Die einheitliche Benutzerschnittstelle, in der IPSEN seine Funktionen dem Entwickler anbietet, findet in einem einheitlichen Implementierungskonzept seine Entsprechung.

Die Implementation der angeführten Werkzeuge des Software-Arbeitsplatzes IPSEN basiert auf graphförmigen Datenstrukturen. Sämtliche Informationen zur Systemstruktur werden im *Systemgraph* gehalten. Jeder ausprogrammierte Programm-Modul wird intern in einem sog. *Modulgraph* repräsentiert. Die Modulgraphen sind am Systemgraph als übergeordnete Datenstruktur angehängt. Informationen zum(r) Projektmanagement/-organisation könnten in einem *Organisationsgraph* abgelegt sein, der mit dem Systemgraph verflochten ist.

Modifikationen an der Systemstruktur oder an einem Programm-Modul bedingen Modifikationen des entsprechenden Graphen. Graph-Modifikationen lassen sich als Ersetzungen von Untergraphen beschreiben. *Graph-Grammatiken* bieten eine geeignete formale Notation für solche Ersetzungen. Da alle internen Datenstrukturen graphförmig sind, sind alle IPSEN-Werkzeuge einheitlich durch Graph-Grammatiken spezifizierbar.

KAPITEL 2: WOHLSTRUKTURIERTE SOFTWARESYSTEME

2.1 PRINZIPIEN DES SYSTEMENTWURFS

Die Tätigkeit des Systementwerfens bezweckt die Fixierung der (statischen) Struktur eines Softwaresystems, das zur Lösung eines in formaler oder informaler Weise vorgegebenen Problems zu entwickeln ist. Ein Algorithmus zur Problemlösung liegt dabei zumindest in ideenhaften Ansätzen vor. Ergebnis des Entwurfsprozesses ist eine (syntaktische) *Systemspezifikation*. Sie umfaßt die Spezifikation der einzelnen Systemkomponenten und ihrer gegenseitigen Beziehungen. Die Notation der Systemspezifikation ist i.a. noch unabhängig von einer bestimmten Implementierungssprache; vielmehr ist die Spezifikation meist in einer speziellen Spezifikationssprache¹⁾ formuliert.

Der Übergang von der "Welt der Ideen" zu einer formalen Formulierung des Entwurfsergebnisses ist Teilaktivität des Programmierens im-Großen (vgl. Abb. 1-1).

Das Ergebnis des Systementwurfs ist für viele späteren Eigenschaften des zu erstellenden Softwaresystems von entscheidender Bedeutung. Man denke dabei an die Transparenz des Systems hinsichtlich der verwendeten Lösungsverfahren, also seine Selbstdokumentierbarkeit, an seine Änderbarkeit, Validierbarkeit (Testbarkeit, Verifizierbarkeit) usw.. Eine aufgrund eines Entwurfsfehlers zu einem späteren Entwicklungszeitpunkt notwendige Korrektur in einem (evtl. komplexen) Softwaresystem äußert sich i.a. in immensem Zeit- und Kostenaufwand.

Es ist daher notwendig, durch eine systematische Vorgehensweise und die Einhaltung gewisser Prinzipien bereits in der Entwurfsphase von vornherein eine überschaubare Struktur in das zu entwickelnde Softwarepaket einzubringen.

Im folgenden seien einige dieser *Prinzipien des Software Engineerings* aufgeführt, denen ein "wohlstrukturiertes" Softwaresystem genügen sollte.

¹⁾ Zeichenketten-Sprachen: MIL /DEKR76/, SPECIAL /LRS79/ u.a., graphische Sprachen: SADT /ROSC77/, HIPO /STA76/ u.a.

2.1.1 MODULARISIERUNG

Ein Softwaresystem sollte *modular* aufgebaut sein.

Es lassen sich zwei Modularisierungsebenen unterscheiden, eine logische und eine physische Modularisierung.

Zunächst soll die *logische* Modularisierung etwas charakterisiert werden. Während des Entwurfsprozesses führen Entwurfsentscheidungen, die hinsichtlich der Einführung von Datenobjekten und Funktionen getroffen werden, direkt zur Spezifikation entsprechender *Systemkomponenten*. Diese Systemkomponenten realisieren die gefällten Entscheidungen¹⁾. Systemkomponenten zeigen insofern Modulverhalten (sind logische Moduln bzw. logische Baustein), als sie Realisierungsdetails dem Restsystem gegenüber verkapseln.²⁾

Die mögliche Kommunikation einer Systemkomponente mit dem Restsystem ist in ihrer *äußeren Schnittstelle* eindeutig geregelt: Sie stellt *Ressourcen* bereit und verwendet selbst Ressourcen anderer Systemkomponenten. Die *Realisierung* der bereitgestellten Ressourcen bleibt nach außen hin verborgen. Sie kann die Einführung lokaler Datenobjekte und Funktionen, also lokale Systemkomponenten notwendig machen (*innere Schnittstelle*). Basierend auf dieser inneren Schnittstelle und unter Verwendung von außerhalb angebotenen (globalen) Ressourcen werden die bereitzustellenden Ressourcen *implementiert*. Implementieren in unserem Sinne umfaßt also ausschließlich das Ausprogrammieren von Ressourcen, wobei bereits während des Systementwurfs spezifiziert wurde, welche anderen lokalen und globalen Ressourcen dazu verwendet werden dürfen.

Es lassen sich *datenorientierte* und *funktionsorientierte* Systemkomponenten unterscheiden.

Zu ersteren zählen Systemkomponenten, die eine verkapselte Datenstruktur als Ressource bereitstellen. Die Manipulation der (nach außen nicht sichtbaren) Datenstruktur ist ausschließlich durch einen Satz ihr zugeordneter *Zugriffsoperationen* möglich. Die Zugriffs-

1) Beispielsweise führt die Entwurfsentscheidung, im Zuge der Syntaxanalyse eines Programms in einer blockstrukturierten Programmiersprache eine Blockliste zu führen, im Compiler zur Spezifikation einer Systemkomponente *BLOCKLISTE*.

2) Für das Restsystem ist es irrelevant, ob die Systemkomponente *BLOCKLISTE* durch eine lineare oder eine verkettete Liste realisiert wird.

operationen sind Bestandteil der äußeren Schnittstelle. Die Datenstruktur selbst wird in der sog. *Repräsentierung* (Bestandteil der inneren Schnittstelle) definiert.¹⁾ Datenorientierte Systemkomponenten besitzen insofern ein Gedächtnis, als die verkapselte Datenstruktur natürlich auch zwischen Aufrufen der Zugriffsoperationen existiert. Der *aktuelle Zustand* einer datenorientierten Systemkomponente wird durch die Ausführung einer Zugriffsoperation i.a. in einen Folgezustand überführt.

Die geschilderte Art von Systemkomponenten ist in der Literatur als *abstraktes Datenobjekt* bekannt /LIZI74/.

Das Prinzip der *Datenabstraktion* erfährt durch *abstrakte Datentypen* /LIZI74/ seine konsequente Erweiterung. Ein abstrakter Datentyp (ebenfalls eine datenorientierte Systemkomponente) stellt als Resource ein "Muster" bzw. eine "Schablone" einer verkapselten und nur über Zugriffsoperationen manipulierbaren Datenstruktur bereit. Abstrakte Datenobjekte, deren Repräsentierung und Satz an Zugriffsoperationen gleich oder zumindest "ähnlich" sind, sind vom gleichen abstrakten Datentyp und können über diesen definiert werden. Während das abstrakte Datenobjekt als Ergebnis einer Abstraktion von einer konkreten Realisierung zu sehen ist, resultiert ein abstrakter Datentyp also aus der Abstraktion von strukturellen "Ähnlichkeiten".

Der Begriff "abstrakter Datentyp" umfaßt in der Literatur auch Aspekte, die die Semantik der möglichen Manipulationen der zugrundgelegten Datenstruktur betreffen (z.B. Beschreibung der Semantik durch die Methode der algebraischen Spezifikation /GUT77/). Von solchen Aspekten wird im folgenden abgesehen.

Ein Softwaresystem, dessen Komponenten gemäß dem angedeuteten Prinzip des *information hidings* /PAR71/ entworfen werden, gewinnt bekanntermaßen an struktureller Überschaubarkeit und Flexibilität hinsichtlich Änderungen bzw. Testmöglichkeiten.

Die funktionsorientierten Systemkomponenten realisieren abstrakte Funktionen. Unter einer *abstrakten Funktion* versteht man einen Algorithmus, der auf abstrakten Datenobjekten (mittels deren Zugriffs-

¹⁾ Würde beispielsweise die Blockliste in unserem Beispiel als lineare Liste mittels eines zweidimensionalen Feldes realisiert werden, würde dieses Feld die Repräsentierung von *BLOCKLISTE* ausmachen.

operationen) operiert und (evtl.) die aktuellen Zustände der Objekte ändert¹⁾ (vgl. auch /LIZI74/).

Im Gegensatz zu abstrakten Datenobjekten ist eine abstrakte Funktion gedächtnislos, da lokale Objekte der Funktion nach deren Ausführung nicht mehr verfügbar sind. Abstrakte Funktionen zeigen also bei gleichem aktuellen Zustand ihrer Argumente gleichbleibendes Ausgabeverhalten, während der Effekt der Zugriffsoperationen eines abstrakten Datenobjekts außer von ihren Eingabewerten auch vom aktuellen Zustand des Objekts abhängt.

Im Gegensatz zur logischen Modularisierung spiegelt die *physische* Modularisierung die Entwurfsentscheidungen als Antwort auf die Frage wider, welche der spezifizierten Systemkomponenten in welchen *Programm-Moduln* der Implementierungssprache (physische Bausteine) realisiert werden sollen. Nicht jede Systemkomponente nämlich wird in einem eigenen Programm-Modul realisiert.

Soll beispielsweise während der Syntaxanalyse eines größeren Programms zu dessen einzelnen Programmeinheiten (Hauptprogramm, Unterprogramme, Moduln) jeweils eine eigene Blockliste geführt werden, ist die Definition eines abstrakten Datentyps *T_BLOCKLISTE* naheliegend. Während die Systemkomponente *T_BLOCKLISTE* dann in einem eigenen Programm-Modul des Compilers eingeführt wird, werden im Compiler-Modul *SYN_ANALYSE* (eigentliche Syntaxanalyse) (dynamisch) entsprechend viele abstrakte Datenobjekte des Typs *T_BLOCKLISTE* kreiert. Den einzelnen dieser Systemkomponenten ist jedoch kein eigener Modul zugeordnet.

Ein Programm-Modul kommuniziert mit anderen Programm-Moduln ausschließlich über eine klar definierte *Export-/Import-Schnittstelle*. Diese Export-/Import-Schnittstelle kann i.a. ebenso wie lokale Vereinbarungen im *Modulrumpf* während des Codierens-im-Großen direkt aus der implementierungssprach-unabhängigen Systemspezifikation abgeleitet werden. Im Zuge des Programmierens-im-Kleinen ist dann der Modulrumpf auszuprogrammieren, wobei Export-/Import-Schnittstelle und lokale Vereinbarungen Vorgaben sind. Programm-Moduln stellen also die Bearbeitungs-Einheiten des Programmierens-im-Kleinen dar.

¹⁾ Vielfach werden in der Literatur Algorithmen, die mehrstellig auf abstrakten Datenobjekten operieren (z.B. Verknüpfungsoperatoren auf bzw. Relationen zwischen abstrakten Datenobjekten) als Zugriffsoperationen des abstrakten Datenobjekts betrachtet, das sich als kartesisches Produkt der Operanden-Objekte ergibt. Die Zugriffsoperationen solcher Produkt-Objekte sind abstrakte Funktionen in unserem Sinne.

Im folgenden Abschnitt werden bewährte Prinzipien erörtert, die Struktur in die potentielle Vielfalt von Beziehungen zwischen Systemkomponenten einbringen.

2.1.2 HIERARCHIEBILDUNG

Ein Softwaresystem sollte *hierarchisch* aufgebaut sein.

Diese aus logischen Moduln (Systemkomponenten) und (wie sich noch zeigen wird) deshalb auch physischen Programm-Moduln gebildete Hierarchie ist das Ergebnis der Anwendung erprobter Entwurfsstrategien wie top-down-, outside-in- bzw. bottom-up-Entwicklung (/DDH72/ bzw. /PAS75/).

Der Prozeß einer sukzessiven Dekomposition einer Problemlösung erscheint "natürlich" und deshalb als Entwurfsstrategie naheliegend.

Bei dieser Vorgehensweise wird zunächst eine Problemlösung auf einer sehr hohen Ebene, einer *abstrakten Maschine*, problemnah formuliert. Diese abstrakte Maschine stellt *abstrakte Datenobjekte* mit entsprechenden manipulierenden Zugriffsoperationen zur Verfügung. Außerdem hält die abstrakte Maschine *abstrakte Funktionen* bereit, die diese abstrakten Datenobjekte lesend, verändernd bzw. vergleichend algorithmisch verknüpfen. Eine solche abstrakte Funktion kann somit als Befehl der abstrakten Maschine gesehen werden. Operanden dieses Befehls sind abstrakte Datenobjekte. Der Systementwerfer löst nun das gegebene Problem, indem er auf dieser abstrakten Ebene den Lösungsalgorithmus in Form einer weiteren abstrakten Funktion formuliert, wobei er die vorgegebenen abstrakten Datenobjekte und Funktionen (Befehle) verwenden kann.

Analog einer durch eine höhere Programmiersprache repräsentierten "konkreten" Maschine (z.B. einer PASCAL-Maschine), die vordefinierte Objekttypen (z.B. integer) anbietet bzw. die Definition neuer Objekttypen (z.B. wochentage) ermöglicht, sieht auch unsere abstrakte Maschine ein Typkonzept vor. Sie stellt Typen abstrakter Datenobjekte, *abstrakte Datentypen* (im Sinne von Abschn. 2.1.1), bereit.

Der Systementwickler entwirft also seine Problemlösung mit Hilfe

von Objekten und Funktionen, die von einer "sehr abstrakten" Maschine AM_n bereitgestellt werden, und spezifiziert sie als Systemkomponenten. Hat er diese auf sehr problemnahem Niveau formulierte Problemlösung als "korrekt" erkannt, beschäftigt er sich mit der Realisierung der verwendeten Abstraktionen, also den verkapselten Datenstrukturen und Algorithmen. Er baut letztere aus Objekten und Funktionen einer "etwas weniger" abstrakten Maschine AM_{n-1} auf, wobei diese Bausteine als Systemkomponenten der nächst tieferen Schicht in der *Systemhierarchie* zu spezifizieren sind. Kurz: eine abstrakte Maschine (AM_n) wird jeweils durch eine "etwas weniger" abstrakte Maschine (AM_{n-1}) realisiert.

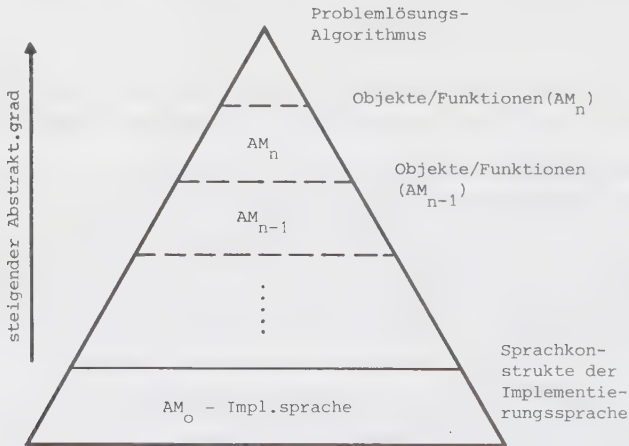


Abb. 2-1

Die hierarchische Systementwicklung terminiert mit dem Erreichen der Ebene der Implementierungssprache. Somit führt die Hierarchie an spezifizierten abstrakten Maschinen AM_i ($0 \leq i \leq n$) mit steigendem Abstraktionsgrad von AM_0 , der Ebene der Implementierungssprache, zu AM_n als der abstrakten Maschine, für die die Problemlösung am problemnächsten formuliert werden konnte (siehe Abb. 2-1).¹⁾

¹⁾ Die dargelegte Sichtweise ist zugegebenermaßen insofern etwas idealisiert, als in praktischen Fällen vielfach die Grenzen zwischen Abstraktionsebenen verwaschen sind bzw. die Realisierung einer Systemkomponente (sinnvollerweise) einige Abstraktionsebenen nach unten hin überspringt.

Im folgenden wird von einer *strengen* Hierarchie ausgegangen, d.h. jedes abstrakte Objekt und jede abstrakte Funktion der abstrakten Maschine AM_{i-1} trägt zur Realisierung genau einer Systemkomponente von AM_i bei. Die derart zwischen Systemkomponenten definierte *Realisiert*-Beziehung impliziert den Begriff des *Teilsystems*: jede Systemkomponente wird durch eine eigene Teilhierarchie, einem Teilsystem, realisiert. Jedes Teilsystem besteht aus einzelnen Systemkomponenten bzw. wiederum Teilsystemen.

Eine hierarchische Systemstruktur wird natürlich auch durch eine bottom-up-Entwicklung (Komposition) erzielt. Dabei werden Abstraktionen (abstrakte Datenobjekte/Datentypen/Funktionen) einer abstrakten Maschine AM_i mit Hilfe von Abstraktionen einer bereits spezifizierten abstrakten Maschine AM_{i-1} definiert. Jede von AM_{i-1} vorgegebene Systemkomponente soll dabei zur Realisierung genau einer Systemkomponente von AM_i beitragen. Somit wird jede neu definierte Abstraktion durch ein hierarchisches Teilsystem von "weniger abstrakten" Systemkomponenten realisiert. Insgesamt werden also bei einer bottom-up-Entwicklung sukzessive hierarchische Teilsysteme zu Teilsystemen einer höheren Hierarchiestufe und schließlich zum Gesamtsystem zusammengesetzt.

Softwaresysteme der Praxis werden weder durch eine reine top-down- noch eine reine bottom-up-Entwicklung, sondern in einer Vorgehensweise mit wechselnden Entwurfsrichtungen entworfen. Trotzdem sollte die Bildung einer strengen Hierarchie als Strukturierungsprinzip befolgt werden.

2.1.3 KONTROLLIERTE INFORMATIONSVERTEILUNG

Es wird später deutlich, daß man bei Softwaresystemen der Praxis mit einer Ressourcen-Nutzung allein aufgrund der Realisiert-Beziehung nicht auskommt. Eine Systemkomponente sollte auch die Ressourcen anderer Systemkomponenten einer begrenzten "Nachbarschaft" benutzen dürfen. Somit wird die Struktur eines Softwaresystems neben der Realisiert-Beziehung auch durch die Beziehung fixiert, die die Benutzung der Systemkomponenten untereinander ausdrückt.

Diese aus der gegenseitigen Benutzung von Systemkomponenten resul-

tierende Beziehung sollte dem Prinzip der *kontrollierten Informationsverteilung* /PAR71/ bzw. der *Zugriffskontrolle* genügen. Dies bedeutet, daß Zugriffe auf Informationen, abstrakte Datenobjekte also, nur nach bestimmten strukturorientierten Regeln vorgenommen werden dürfen und stets transparent sein müssen. Diese Regeln begrenzen exakt den Systemkontext, in dem ein abstraktes Datenobjekt für andere Systemkomponenten zugreifbar ist.

Derartige Regeln haben folgende Überlegungen zu berücksichtigen:

- Die Realisierung einer Systemkomponente darf nach außen hin nicht sichtbar sein (information hiding). Da eine Systemkomponente i.a. durch ein ganzes Teilsystem realisiert wird, dürfen auch sämtliche Systemkomponenten dieses Teilsystems von außen nicht erreichbar sein.
- Beispiele der Praxis zeigen, daß für eine Systemkomponente in geregeltem Rahmen hierarchiemäßig höher angesiedelte abstrakte Datenobjekte zugreifbar sein sollten. Wir werden in einem späteren Abschnitt ein derartiges Beispiel anführen.
- Sämtliche möglichen Zugriffe auf abstrakte Datenobjekte sind explizit zu spezifizieren. Außerdem sollte jede Systemkomponente Kenntnis darüber besitzen, welche Zugriffe aus dem Teilsystem, durch das sie realisiert wird, heraus maximal möglich sind. Durch beide Maßnahmen wird die Zugriffstransparenz erhöht, die für die Entwicklung sicherer Systeme unerlässlich ist.

2.2 STRUKTURMODELL

In den letzten Abschnitten wurden bewährte Prinzipien des Software-Engineerings aufgeführt. In diesem Abschnitt werden wir konkret ein *Strukturmodell* für hierarchische und modulare ("wohlstrukturierte") Softwaresysteme einführen. Die Elemente dieses konzeptuellen Modells haben ihre Motivation in den obenerwähnten Prinzipien. Das Strukturmodell repräsentiert also einen "Baukasten", der ein Softwaresystem gemäß den Erkenntnissen des modernen Software-Engineerings zu spezifizieren und zu implementieren gestattet. Es ist als Weiterentwicklung eines in /ALT78/ vorgeschlagenen Modells zu sehen.

Bei der folgenden Diskussion der einzelnen Strukturmodell-Elemente wird stets auch ihre Formulierung in einer Spezifikationssprache betrachtet, die speziell für den vorgestellten Software-Arbeitsplatz entworfen wurde.¹⁾ Außerdem wird die korrespondierende Darstellung im Systemgraph, der die Struktur eines zu entwickelnden Software-systems (also die kontextfreie und kontextsensitive Syntax der Systemspezifikation) repräsentiert, angegeben. Formal wird der System-graph erst in einem späteren Abschnitt definiert.

Zur Einführung der Strukturelemente dient ein Softwaresystem aus der Praxis, *ASSMEM*, als erläuterndes Beispiel. Dieses System wird zunächst im folgenden Einschub vorgestellt.

2.2.1 ASSMEM - EIN SOFTWARE-ASSOZIATIVSPEICHER

ASSMEM /GAN81/ ist ein softwaremäßig implementierter Assoziativspeicher, der es gestattet, Dateneinheiten und zwischen ihnen bestehende binäre Relationen derart abzuspeichern, daß assoziative Fragen effizient beantwortet werden können.²⁾ Dieses Softwaresystem wurde gegenüber seinem Vorläufer *DATAS* /ENW74/ hinsichtlich Konzept und Implementation verbessert.

ASSMEM sieht drei Arten von Dateneinheiten vor:

Single-Value-Entities (SVE's),
Multi-Value-Entities (MVE's) und
Relation-Entities (RE's).

Die SVE's stellen die *Grundeinheiten*, die MVE's *Mengen* von Dateneinheiten aller drei Arten und RE's eine Menge von geordneten Tripeln (REL, ITEM1, ITEM2), *Triaden* genannt, dar. Eine Triade (REL, ITEM1, ITEM2) drückt aus, daß ITEM1 und ITEM2 in der Relation REL zueinander stehen. Somit sind RE's nichts anderes als zweistellige *Relationen*.³⁾

ASSMEM bietet nun dem Benutzer in etwa folgende Schnittstelle zur Manipulation der Dateneinheiten und für assoziative Fragen an:

1) Die gewählte Syntax ist natürlich beliebig austauschbar.

2) Es lassen sich also sämtliche Sachverhalte abspeichern, die durch Graphen repräsentiert werden können. Die Ablage erfolgt unabhängig von der speziellen Graphenstruktur und der Größe des Graphen.

3) ITEM1 und ITEM2 in einer Triade können von beliebiger Art, also u.a. auch selbst Relationen sein.

- Auswechseln des externen, vom Benutzer eingeführten Namens
- Zuordnen und Löschen eines Synonyms zu einer Entity
- Erzeugen und Löschen einer Entity
- Ausgeben aller Synonyme einer Entity
- Abspeichern einer Entity als Element einer MVE
- Test auf Elementbeziehung
- Löschen einer Elementbeziehung
- Abspeichern einer Triade zu einer RE
- Löschen einer Triade
- assoziative Fragen der Formen

(REL,ITEM1,ITEM2), (REL,ITEM1,?), (REL,?,?),
wobei das Fragezeichen die Stelle charakterisiert, deren Werte erfragt werden sollen.

Auf die erste Frage ergibt sich als Antwort true oder false, je nachdem, ob der assoziative Speicher eine Triade (REL,ITEM1,ITEM2) enthält oder nicht.

Als Antwort auf die nächste Frage wird eine MVE bereitgestellt, die als Elemente alle Entities Y enthält, so daß (REL,ITEM1,Y) eine auffindbare Triade ist.

Die letzte Frage liefert als Ergebnis die Menge aller Paare von Entities X und Y, so daß (REL,X,Y) eine abgespeicherte Triade ist.

Analog sind auch die assoziativen Fragen (REL,?,ITEM2), (?,?,ITEM2), (?,?,ITEM1,?), (?,?,ITEM2) möglich.

2.2.2 SYSTEMKOMPONENTEN

Im folgenden werden wir die *Systemkomponenten* als die spezifikations-sprachlichen Objekte einführen.

Es sind mehrere *Arten* von Systemkomponenten vorgesehen. Diese ergeben sich direkt aus dem Wunsch heraus, die aus dem Entwurfsprozeß resultierende Problemlösung in möglichst durchgängiger Weise in der Spezifikationssprache spezifizieren zu können.

In Abschnitt 2.1.2 wurde der Entwurfsprozeß basierend auf dem Entwurfskonzept der hierarchisch angeordneten abstrakten Maschinen ausgeführt. Jede abstrakte Maschine läßt sich über abstrakte Datenobjekte/-typen und Funktionen der jeweils nächst niedrigeren Maschine programmieren. Diese Arten an logischen Bausteinen sind aufgrund der Forderung nach Durchgängigkeit in unserem Ansatz deshalb auch direkt spezifikationssprachliche Objekte, also Systemkomponenten.¹⁾

¹⁾ /ALT78/ schlägt zum Aufbau modularer Softwaresysteme sog. Datenmoduln (D-Moduln) und funktionsorientierte Moduln (F-Moduln) als spezifikationssprachliche Objekte vor. Auf Unterschiede zu den abstrakten Datenobjekten und abstrakten Funktionen unseres Ansatzes wird an entsprechender Stelle hingewiesen.

2.2.2.1 ABSTRAKTES DATENOBJEKT

Unter einem *abstrakten Datenobjekt* (kurz ADO) versteht man - wie in den Abschnitten 2.1.1 und 2.1.2 angerissen - ein (meist "komplexer" strukturiertes) Datenobjekt einer höheren Abstraktionsebene. Es realisiert eine nach außen verborgene Entwurfsentscheidung ("verkapselte Datenstruktur") und ist nur in kontrollierter Weise lesend und/oder schreibend *zugreifbar*, nämlich über einen Satz von *Zugriffsoperationen*.

Beispiel:

ASSMEM ist eine komplexe Datenstruktur, die nur über die Operationen der *ASSMEM*-Benutzerschnittstelle manipulierbar ist. Details der *ASSMEM*-Realisierung sind für den Benutzer nicht von Interesse und werden vor ihm verborgen. *ASSMEM* stellt also ein abstraktes Datenobjekt dar.

Ein abstraktes Datenobjekt $ADO^{1)}$ wird als Systemkomponente der Art *ado* spezifiziert. Seine Zugriffsoperationen OPI werden in der *äußeren Schnittstelle* der Systemkomponente bekanntgemacht:

```
provides
  op OP1(..., F_PAR1j : TYPE1j, ...);
  ⋮
  op OPk(..., F_PARKj : TYPEKj, ...);
```

ADO wird durch abstrakte Datenobjekte/Funktionen der nächst tieferen abstrakten Maschine realisiert. Die *innere Schnittstelle* von *ADO* spiegelt die Entwurfsentscheidungen wider, die zur Definition des *ADO*'s durch Verfeinerung bzw. Zusammensetzung getroffen wurden.

Die innere Schnittstelle legt somit zum einen die Datenstruktur (*Repräsentierung*) fest, die dem zu realisierenden *ADO* zugrundeliegt:

```
representation
  REPRKOMPI
  ⋮
  REPRKOMPL
```

Die Repräsentierungs-Komponenten $REPRKOMPI$ sind i.a. wieder abstrakte Datenobjekte, nämlich Objekte der abstrakten Maschine, die *ADO*

¹⁾ Im weiteren werden Bezeichner von Systemkomponenten *kursiv* geschrieben.

realisiert.

Auf entsprechend niedriger Abstraktionsebene kann es sich jedoch dabei auch um Datenobjekte der Implementierungssprache (*einfache Datenobjekte*, kurz EDO'e) handeln. Die Typen dieser Datenobjekte könnten etwa den in einer PASCAL-ähnlichen Programmiersprache definierbaren Datentypen (*einfache Datentypen*, kurz EDT'en) entsprechen. In der Systemspezifikation treten diese Datentypen jedoch zunächst nur angewandt auf: Die entsprechenden Typdefinitionen werden erst in der Implementierungsphase eingebracht.

Beispiel (Repräsentierung eines ADO's *STACK*):

representation

```
edo STACK_STRUCT : array[1:20] of T_ELEM1);  
edo TOP_POINTER : T_ELEM_PT;
```

Die einem ADO *ADO* zugrundegelegte Datenstruktur besteht "in erster Näherung" aus dem einfachen Verbund der Repräsentierungs-Komponenten²⁾. Diese Datenobjekte besitzen nur innerhalb von *ADO* (also "örtliche") Bedeutung, können also z.B. nicht zur Realisierung anderer abstr. Datenobjekte herangezogen werden (streng hierarchische Realisiert-Beziehung).

Die spätere Implementierung der Zugriffsoperationen *OPI* kann die Benutzung von (Bündeln von) abstrakten Funktionen *LOC_FUNCj* nahelegen, die ebenfalls nur lokal in *ADO* von Bedeutung sind:

local functions

```
func LOC_FUNC1(...);  
      ⋮  
func LOC_FUNCm(...);
```

Jede dieser Funktionen *LOC_FUNCj* operiert auf (evtl. mehreren) Komponenten der Repräsentierung.

1) Typ der Kellerelemente

2) Es interessiert zunächst noch keine detailliertere Strukturierung. Zugriffe der Repräsentierungs-Komponenten untereinander bleiben momentan ebenfalls noch unspezifiziert.

Beispiel:

Die äußere Schnittstelle des Softwaresystems *ASSMEM*, also seine Benutzerschnittstelle, stellt eine Reihe von Operationen zur Manipulation der Dateneinheiten bereit:

```
SYSBEGIN          /* System-Initialisierung */
DECLARE_SVE       /* Erzeugen einer SVE */
DECLARE_MVE       /* Erzeugen einer MVE */
:
STORE_TRIADE      /* Ablegen einer Triade */
:
SYSEND           /* System-Ende */
```

Der Entwurf von *ASSMEM* sieht eine Zerlegung der assoziativen Datenstruktur in drei getrennte Speicherbereiche vor, den sog. Namenspeicher, Datenspeicher und Triadenspeicher.

Der Namenspeicher setzt die vom Benutzer gewählten externen Namen der Dateneinheiten in (leichter handhabbare, da kürzere) interne um.

Der Datenspeicher dient der Ablage von Dateneinheiten einschl. Information zu ihrer Verwaltung.

Im Triadenspeicher sind die Beziehungen zwischen den Dateneinheiten in Listenstrukturen abgelegt, die eine relativ effiziente Beantwortung der assoziativen Fragen ermöglichen.

Diese drei Speicherbereiche können wegen ihrer Größe nicht komplett im Hauptspeicher resident gehalten, sondern müssen auf Platte ausgelagert werden. Sie sind deshalb in Seiten einer festen, leicht transferierbaren Größe unterteilt. Die Ein-/Auslagerung der Seiten wird von einem system-eigenen Pagingssystem verwaltet. Eine Pagetabelle führt die dazu nötige Verwaltungsinformation.

Wie in /GAL78/ beschrieben, lassen sich die drei Speicherbereiche und die Pagetabelle als abstrakte Datenobjekte (*NM*, *DM*, *TM*, *PL*) betrachten, die zur Repräsentierung von *ASSMEM* gehören. Zur Übertragung von Informationen zwischen den Speicherbereichen ist ein Datenpuffer nötig, auf den alle Speicherbereiche zugreifen. Er ist ebenfalls als abstraktes Datenobjekt (*CBUF*) der Repräsentierung zu sehen.

Es erwies sich auch als sinnvoll, zur Implementierung der Zugriffsoptionen der *ASSMEM*-Benutzerschnittstelle einige lokale abstrakte Funktionen (*RELEASE_ENT*, ...) bereitzustellen, die mehrstellig auf den drei Speicherbereichen operieren. Es handelt sich dabei um Aktionsfolgen, die in mehreren Zugriffsoperationen in gleicher Weise durchzuführen sind und deshalb aus Effizienz- und Übersichtlichkeitsgründen besser als eigene abstrakte Funktionen anzubieten sind.

ASSMEM könnte wie in Abb. 2-2 spezifiziert werden.

```
ado ASSMEM

  external interface

    provides
      op SYSBEGIN(...);
      op DECLARE_SVE(...);
      op DECLARE_MVE(...);
      :
      op STORE_TRIADE(...);
      :
      op SYSEND(...);

      (in-)formal object specification1)
      ....

  end external interface

  ado-realization REALIS12)

    internal interface

      representation
        ado NM : ...;
        ado DM : ...;
        ado TM : ...;
        ado PL : ...;
        ado CBUF : ...;

      local data types3)
      —

      local functions
        func RELEASE_ENT(...);
        :

    end internal interface

  end ado-realization REALIS1

end ado ASSMEM
```

Abb. 2-2: abstraktes Datenobjekt *ASSMEM*

- 1) Hier kann die Semantik von *ASSMEM* (z.B. axiomatisch) spezifiziert werden. Diese Angabe wird jedoch vom vorgestellten Software-Arbeitsplatz weder in irgendeiner Form überprüft noch ausgewertet, sondern nur als Kommentar betrachtet.
- 2) Der Zweck dieses Bezeichners wird später klar.
- 3) Auf diese Angabe kommen wir im folgenden Abschnitt zu sprechen.

Die Spezifikation des abstrakten Datenobjekts *ASSMEM* wird im Systemgraph, der ein zu entwickelndes Softwaresystem repräsentiert, wie folgt dargestellt:

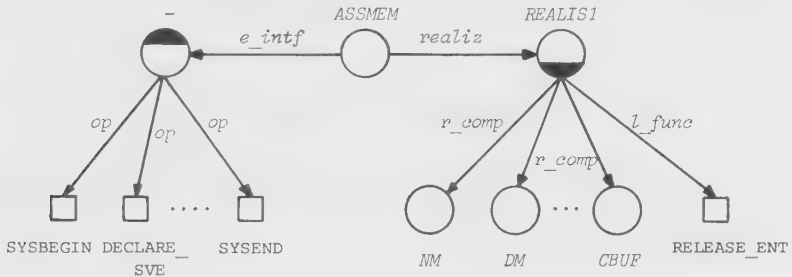


Abb. 2-3

Prinzipiell trägt jeder Knoten des Systemgraphen zwei Markierungen: den Bezeichner und die Art des Spezifikationselements, das der Knoten repräsentiert. Aus Lesbarkeitsgründen werden jedoch die Werte der zweiten Markierung durch unterschiedliche Knotenformen ausgedrückt.

Der Knoten *ASSMEM* steht für die gesamte Systemkomponente *ASSMEM*



(die Systemkomponenten-Art *ado* wird durch den geschlossenen Kreis als Knotenform beschrieben).

Der Zielknoten der Kante mit Markierung *e_intf* (external interface) repräsentiert die äußere Schnittstelle von *ASSMEM*. Diese besitzt keinen eigenen Bezeichner (Markierung "-"). Die Tatsache, daß sie die Schnittstelle eines ADO's darstellt, wird durch die Knotenform für ein ADO ausgedrückt, die durch eine Kennung für "Kommunikation mit der Umgebung" (insgesamt also ) modifiziert ist.¹⁾

An diesem Knoten sind über Kanten mit Markierung *op* (operation) die Knoten für die *ASSMEM*-Operationen angehängt.²⁾

Die Kante mit Markierung *realiz* (realization) verweist auf den Knoten, der für die mit *REALIS1* bezeichnete Realisierung steht.³⁾

1) Prinzipiell ist die Kennzeichnung, daß der Knoten die äußere Schnittstelle eines ADO's repräsentiert, unnötig, da diese Information auch dem mit *ASSMEM* bezeichneten Knoten entnommen werden kann. Sie erhöht jedoch die Lesbarkeit späterer Systemgraph-Ausschnitte.

2) Die Operationen sind nicht ausschließlich Gegenstand des Programmierens-im-Großen, sondern (während der Implementierungsphase) auch des Programmierens-im-Kleinen. Die entsprechenden Knotenformen werden deshalb wie für alle Größen, die keine Systemkomponenten sind (einfache Datenobjekte/-typen), etwas verkleinert gezeichnet.

3) Die Notwendigkeit eines eigenen Knotens für die Realisierung wird später noch deutlich.

Das schwarze Scheibendrittel der Knotenform assoziiere die innere Schnittstelle.

Die Knoten für die Repräsentierungs-Komponenten *NM*, *DM*, *TM*, *PL* und *CBUF* sind Zielknoten von *r_comp*-Kanten (representation-component), die vom Realisierungsknoten ausgehen. Die *l_func*-Kante (local function) und der entsprechende Zielknoten halten die lokale abstrakte Funktion *RELEASE_ENT* fest.

2.2.2.2 (GENERISCHER) ABSTRAKTER DATENTYP

Jedes abstrakte Datenobjekt ist von einem bestimmten *Typ*. Dieser ist durch die Repräsentierung des Objekts und die ihm zugeordneten Operationen festgelegt.

Wie viele praktische Beispiele zeigen, lassen sich oft mehrere Repräsentierungs-Komponenten eines ADO's insofern gemeinsam als eine Klasse von abstrakten Datenobjekten definieren, als ihr interner struktureller Aufbau (Repräsentierung) und die (syntaktische und semantische) Spezifikation ihrer Operationen *gleich* oder zumindest "*ähnlich*" ist. Die betreffenden abstrakten Datenobjekte sind dann Objekte vom *gleichen Typ*, nämlich vom gleichen *abstrakten Datentyp* (kurz ADT) oder von einem *generischen abstrakten Datentyp* (kurz GADT).

Ein ADT kann als "Schablone" für ein abstraktes Datenobjekt gesehen werden. Ein GADT dagegen ist ein ADT, dessen Definition (nur) bis auf gewisse Größen, den *generischen Parametern* nämlich, festliegt ("Typgenerator")¹⁾. Ein GADT ist somit eine "Schablone" für einen ADT, also eine "Meta-Schablone".

Treten (generische) abstrakte Datentypen in Objektdeklarationen angewandt auf, werden sie *benutzt*.

Zur Spezifikation (generischer) abstrakter Datentypen stellt unser Strukturmodell Systemkomponenten der Arten *adt* bzw. *gadt* bereit.

Zunächst sind zwei Möglichkeiten vorgesehen, ein Objekt *ADO* des Typs *ADT* zu deklarieren:

-
- 1) Standardbeispiel ist der Typ eines allgemeinen Kellers, wobei der Typ der Kellerelemente als generischer Parameter noch offen gelassen ist.
 - 2) Wird später noch präzisiert.

- a) Die Definition des Typs wird - wie in Abb. 2-2 - direkt in die Deklaration des ADO's eingebracht (*ADO-Deklaration mittels implizitem ADT*)¹⁾. Diese Vorgehensweise ist sicher dann zweckmäßig, wenn der Typ des ADO's in keiner weiteren ADO-Deklaration benötigt wird.
- b) Der Typ wird in einer eigenen Typdeklaration unter dem Bezeichner *ADT* eingeführt (*ADT-Deklaration mittels Typdefinition*). Nun können abstrakte Datenobjekte durch Bezugnahme auf diesen Typbezeichner deklariert werden (*ADO-Deklaration mittels explizitem ADT*)²⁾. Dabei kann eine *Abmagerung* des Operationensatzes vorgenommen werden, so daß das deklarierte ADO nur eine Teilmenge der *ADT*-Operationen als eigene Operationen übertragen bekommt. Dies ist vor allem im Zusammenhang mit abstrakten Datentypen sinnvoll, die aufgrund ihres Charakters allgemeiner Verwendbarkeit im System mehrfach benötigt werden. Die *Abmagerung* des Operationensatzes bietet dann die Möglichkeit einer Anpassung an die tatsächlichen lokalen Notwendigkeiten.

Beispiel:

ASSMEM als Einbenutzersystem ist zunächst als abstraktes Datenobjekt zu sehen (vgl. Abschnitt 2.2.2.1³⁾). Wollen jedoch mehrere Benutzer jeweils einen eigenen Assoziativspeicher verwenden, sind entsprechend viele Inkarnationen der *ASSMEM*-Repräsentierung bereitzustellen. Der (wiedereintrittsfähige) Code der *ASSMEM*-Operationen soll jedoch nur einfach existieren. Der Assoziativspeicher ist somit als abstrakter Datentyp *T_ASSMEM* zu spezifizieren und implementieren⁴⁾. Anschließend können mehrere *T_ASSMEM*-Objekte *ASSMEMi* (evtl. unter *Abmagerung* der Operationenmenge) deklariert werden (*ADO-Deklaration mittels explizitem ADT*):

1) Analog einer Objektdeklaration im Bereich des Programmierens-im-Kleinen:
" VERBUND : record ... end record; "

2) Analogie zum Programmieren-im-Kleinen:
" VERB_TYP : record ... end record;
VERBUND : VERB_TYP; "

3) Dort wurde *ASSMEM* in einer ADO-Deklaration mittels implizitem ADT eingeführt.

4) Die Repräsentierung von *T_ASSMEM* (bzw. jede ihrer Komponenten) hat also nur formalen Charakter.

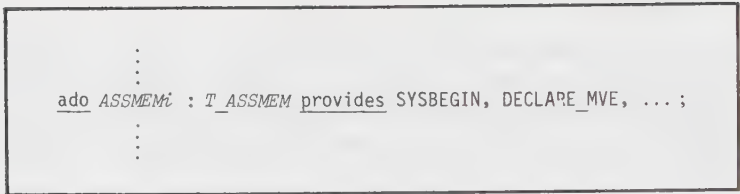


Abb. 2-4

Darstellung im Systemgraph:

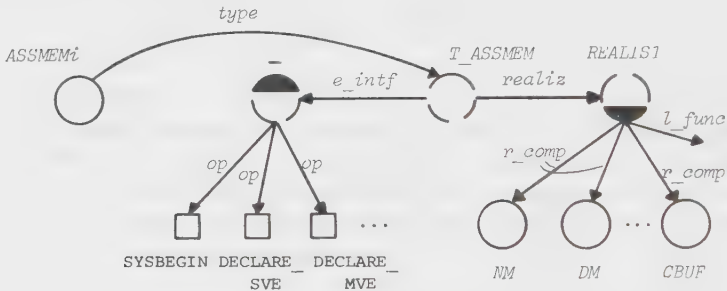


Abb. 2-5

Die Darstellung des ADT's T_ASSMEM weicht von der des ADO's $ASSMEM$ nur in der Knotenmarkierung ab, die die Art der Systemkomponente ausdrückt. Die Markierung "adt" des zusätzlich mit T_ASSMEM markierten Knotens wird durch einen durchbrochenen Kreis als Knotenform¹⁾ repräsentiert. Diese Kreisform überträgt sich auch auf die graphische Repräsentierung der Knoten für die äußere Schnittstelle und die Realisierung von T_ASSMEM . Die Beziehung " $ASSMEMi$ ist vom expliziten Typ T_ASSMEM " wird durch die *type*-Kante vom Knoten mit Markierung $ASSMEMi$ (Knoten repräsentiert ein ADO) zum Knoten mit Markierung T_ASSMEM ausgedrückt.

Nun gilt es noch die Abmagerung des T_ASSMEM -Operationensatzes zu berücksichtigen. Dabei wird von der Idee ausgegangen, dem ADO $ASSMEM$ ebenfalls eine äußere Schnittstelle zu übertragen, die zwar nicht explizit als solche in der Systemspezifikation auftritt, sich jedoch implizit aus der Deklaration von $ASSMEMi$ ergibt.²⁾ Wir nennen diese Art einer äußeren Schnittstelle eine *virtuelle äußere Schnittstelle*. Die virtuelle äußere Schnitt-

- 1) Ein ADT hat den Charakter einer Rahmenvorgabe für ein ADO.
- 2) Jede Systemkomponente besitzt eine äußere Schnittstelle.

stelle von $ASSMEM_i$ enthält dann die T_ASSMEM -Operationen, die $ASSMEM_i$ eingeräumt werden.

Wir wollen virtuelle äußere Schnittstellen auch im Systemgraph berücksichtigen, da dies später einige Konsistenzprüfungen erleichtern wird. Der Systemgraph-Ausschnitt von Abb. 2-5 ist deshalb folgendermaßen zu ergänzen:

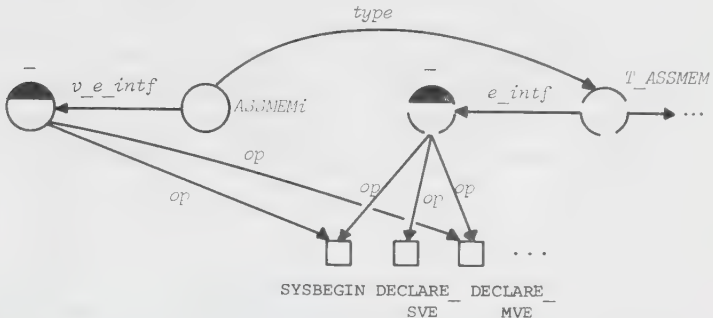


Abb. 2-6

Der mit $ASSMEM_i$ markierte Knoten wird also über eine $v_e__intf$ -Kante (virtual external interface) mit dem Knoten für die virtuelle äußere Schnittstelle verbunden. Von diesem Knoten aus verweisen op -Kanten auf die T_ASSMEM -Operationen, die auch $ASSMEM_i$ übertragen wurden.

Analog der virtuellen äußeren Schnittstelle besitzt $ASSMEM_i$ auch eine virtuelle innere Schnittstelle, nämlich eine Inkarnation der inneren Schnittstelle von T_ASSMEM . Im Systemgraph bleiben jedoch virtuelle innere Schnittstellen unberücksichtigt.

Ein generischer abstrakter Datentyp ist im Unterschied zu einem gewöhnlichen abstrakten Datentyp noch mit verschiedenen Kreierungsparametern (*generische Parameter*)¹⁾ versehen, so daß er eine ganze Klasse von abstrakten Datentypen repräsentiert. Die abstrakten Datentypen dieser Klasse besitzen Repräsentierungen, die bis auf die von den generischen Parametern abhängigen Strukturierungsdetails strukturell gleich sind, und gemeinsame Zugriffsoperationen mit gleichem Ein-/Ausgabe-Verhalten.

Durch eine (evtl. stufenweise) aktuelle Parametrisierung eines GADT's (*Konkretisierung*) werden abstrakte Datentypen als *Ausprägungen*

¹⁾ Bei diesen Parametern handelt es sich um Entwurfszeit-Parameter. Die verschiedenen Parametrisierungsmöglichkeiten werden in einem späteren Abschnitt detaillierter diskutiert.

des GADT's gewonnen. Ein ADT kann also außer durch direkte Angabe seiner Definition (*ADT-Deklaration mittels Typdefinition*) auch dadurch deklariert werden, daß auf einen GADT mit vollständiger Aktualparametrisierung Bezug genommen wird (*ADT-Deklaration mittels GADT*). Gleichermäßen ist auch die Deklaration eines ADO's vorgesehen, in der man sich direkt auf einen GADT mit vollständiger Aktualparametrisierung bezieht (*ADO-Deklaration mittels GADT*).

Ein GADT selbst kann dadurch deklariert werden, daß die entsprechende Typdefinition angegeben wird (*GADT-Deklaration mittels Typdefinition*) oder er durch partielle Konkretisierung¹⁾ als *partielle Ausprägung* eines anderen GADT's eingeführt wird (*GADT-Deklaration mittels GADT*).²⁾

Beispiel:

Jeder Benutzer unseres Assoziativspeichers besitzt die Möglichkeit, die Datenstruktur des Assoziativspeichers den Erfordernissen seiner speziellen Anwendung anzupassen. Dies betrifft vor allem die Dimensionierung der drei *T_ASSMEM*-Speicherbereiche *NM*, *DM* und *TM*.

Der ADT *T_ASSMEM* ist also noch mit formalen generischen Parametern versehen, sodaß erst die entsprechenden Aktualparameter die zugrundeliegende Repräsentierung vollständig definieren: Es liegt strenggenommen ein GADT *GT_ASSMEM* vor. Angenommen, *GT_ASSMEM* besitzt drei formale generische Parameter (die Anzahl der Namen-, Daten- und Triadenspeicher-Seiten), so würde seine Deklaration mittels Typdefinition lauten:

```
gadt GT_ASSMEM (edo NM_SIZE, DM_SIZE, TM_SIZE : INTEGER)  
    external interface  
        :  
        :  
end_gadt GT_ASSMEM
```

Abb. 2-7

1) Einige (jedoch nicht alle) Formalparameter werden durch aktuelle ersetzt.

2) /ALT78/ führt wohl obige Deklarationsmöglichkeiten an. In seiner formalen Beschreibung modularer Programmsysteme bleiben jedoch detailliertere Aspekte hinsichtlich der Deklaration von (generischen) abstrakten Datentypen unberücksichtigt.

Darstellung im Systemgraph:

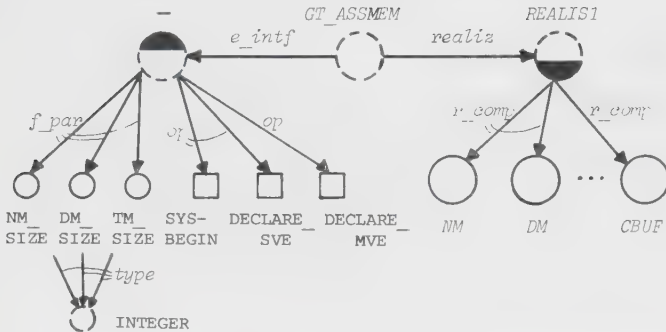


Abb. 2-8

Gegenüber der Darstellung eines ADT's ergeben sich für die Darstellung eines GADT's zwei Änderungen. Zum einen ist die Grundform der Knoten ein stark durchbrochener Kreis.¹⁾ Zum anderen werden über Kanten mit der Markierung *f_par* (formal parameter) Knoten für die formalen generischen Parameter an den Knoten der äußeren Schnittstelle angehängt.

Durch Konkretisierung von *GT_ASSMEM* läßt sich nun z.B. ein ADT *T_SMALL_ASSMEM* als Ausprägung deklarieren (ADT-Deklaration mittels GADT), der den Typ eines Assoziativspeichers mit Minimalkapazität darstellt. Der Operationensatz von *T_SMALL_ASSMEM* kann gegenüber dem von *GT_ASSMEM* eingeschränkt werden:

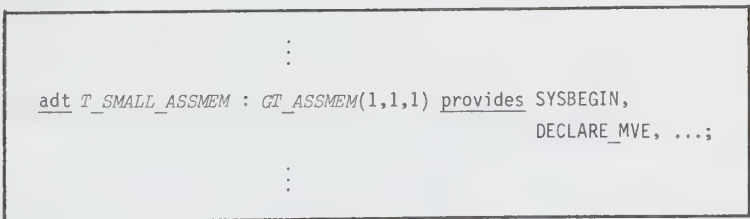


Abb. 2-9

¹⁾ Ein GADT stellt die Schablone zu einer Schablone (ADT) dar.

Aus Orthogonalitätsgründen wird bei der Darstellung im Systemgraph zunächst ein unbenannter, explizit nicht benutzbarer ADT als entsprechende Ausprägung von *GT_ASSMEM* eingebracht. Die Einführung von *SMALL_ASSMEM* wird dann als Deklaration mittels dieses "Phantasie"-ADT's aufgefaßt, so daß sich im Systemgraph folgender Zusammenhang ergibt:

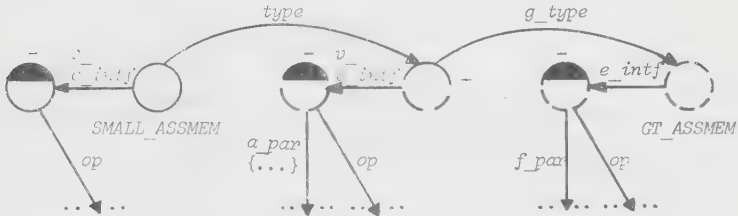


Abb. 2-12

Die an den "Phantasie"-ADT übertragene virtuelle äußere Schnittstelle ergibt sich wie bei einer ADT-Deklaration mittels GADT. Die virtuelle äußere Schnittstelle von *SMALL_ASSMEM* umfaßt nur die Operationen, die diesem Objekt vom "Phantasie"-ADT übertragen werden, nicht jedoch die aktuellen generischen Parameter der Typkonkretisierung.

Analog einer ADT-Deklaration mittels GADT läßt sich auch ein GADT deklarieren, indem nur eine echte Teilmenge der formalen generischen Parameter durch aktuelle abgesättigt wird. Beispielsweise könnte ein generischer Typ "Assoziativspeicher" benötigt werden, in dem die Größe des Namenspeichers bereits fixiert ist:

```

:
:
gadt GT_ASSMEM_WITH_SMALL_NM (edo DM_SIZE, TM_SIZE : INTEGER):
    GT_ASSMEM (1, DM_SIZE, TM_SIZE) provides DECLARE_MVE,...;
:
:

```

Abb. 2-13

GT_ASSMEM_WITH_SMALL_NM besitzt im Gegensatz zu *GT_ASSMEM* nur noch zwei formale generische Parameter.¹⁾

Darstellung im Systemgraph:

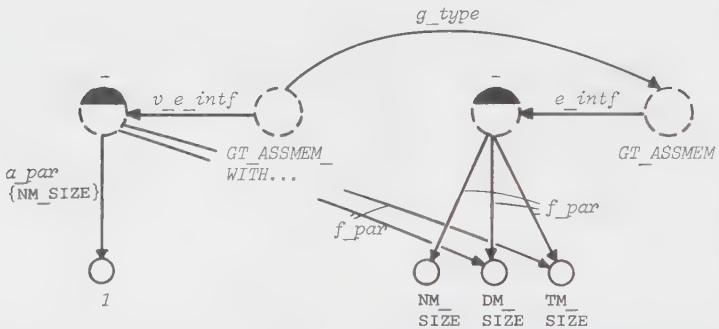


Abb. 2-14

Neben den Kanten für die aktuellen generischen Parameter werden vom Knoten der virtuellen äußeren Schnittstelle von *GT_ASSMEM_WITH_SMALL_NM* *f_par*-Kanten zu den Knoten für die formalen generischen Parameter gezogen, die von *GT_ASSMEM* übernommen werden.

(Generische) abstrakte Datentypen können lokal in der inneren Schnittstelle von abstrakten Datenobjekten bzw. wiederum von (generischen) abstrakten Datentypen beispielsweise zur Deklaration von Repräsentierungs-Komponenten benötigt werden. Derart nur örtlich nötige (generische) abstrakte Datentypen *LOC_ADT_i* bzw. *LOC_GADT_j* werden in der inneren Schnittstelle der entsprechenden Vater-Systemkomponente unter local data types eingeführt.

Im gleichen Spezifikationsabschnitt können auch einfache Datentypen deklariert werden, die beispielsweise zur Deklaration einfacher Datenobjekte in der Repräsentierung oder zur Spezifikation formaler generischer Parameter benötigt werden. Die Definition dieser Daten-

¹⁾ Eine Umbenennung dieser formalen generischen Parameter gegenüber den korrespondierenden formalen Parametern des konkretisierten GADT's ist prinzipiell möglich, wollen wir jedoch nicht vorsehen.

typen kann jedoch, da ihre Syntax stark implementierungssprach-abhängig ist, erst zur Implementierungszeit nachgetragen werden:

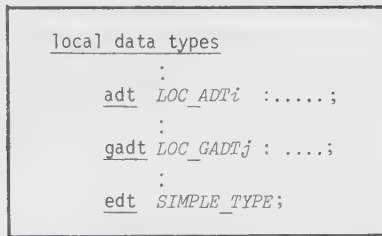


Abb. 2-15

Entsprechende Darstellung im Systemgraph:

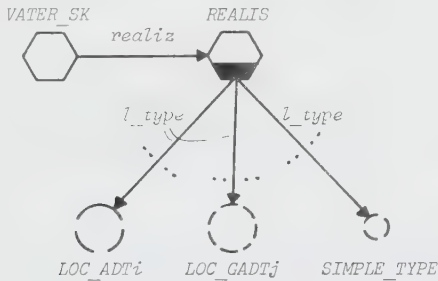


Abb. 2-16

Ist die Art einer Systemkomponente bei der Darstellung eines bestimmten Sachverhalts bedeutungslos, wird der Knoten für diese Systemkomponente als  gezeichnet. Die Kantenmarkierung *l_type* steht für local type.

2.2.2.3 TYPKONZEPT

Die vorgestellte Spezifikationssprache sieht eine *strenge Typisierung* vor: Ein abstraktes Datenobjekt, das als Objekt eines bestimmten abstrakten Datentyps deklariert wurde, kann nur Werte dieses ADT's annehmen bzw. es dürfen nur Operationen dieses ADT's auf das Objekt angewandt werden. Ein solches Typkonzept fördert die Lesbarkeit und Wartbarkeit, in erster Linie aber die Sicherheit des spezifizierten Systems: Bereits zur Entwurfszeit läßt sich jede inkonsistente Verwendung eines ADO's entdecken.

Zwei abstrakte Datenobjekte *ADO1* und *ADO2* sind vom *gleichen* ADT, also *typäquivalent*, wenn in ihrer Deklaration auf dieselbe Typdefinition Bezug genommen wird.

Beispiele (*STACK1* und *STACK2* sind typäquivalent):

- a) ado *STACK1*, *STACK2* : *T_STACK*;
- b) ado *STACK1*, *STACK2* : *GT_STACK*(*INTEGER*);

Typäquivalenz ist also nicht durch ähnliche oder gleiche Repräsentierung der Objekte bedingt: Selbst wenn zwei Typdefinitionen textuell völlig übereinstimmen, führen sie jeweils einen neuen Typ ein.¹⁾

Beispiele (*STACK1* und *STACK2* sind nicht typäquivalent):

- a) ado *STACK1* : *GT_STACK* (*INTEGER*);
ado *STACK2* : *GT_STACK* (*INTEGER*);
- b) adt *T_INT_STACK* : *GT_STACK* (*INTEGER*);
ado *STACK1* : *T_INT_STACK*;
ado *STACK2* : *GT_STACK* (*INTEGER*);

Dieses Typkonzept²⁾ hat neben einer höheren Transparenz der Systemspezifikation den Vorteil einer vollständigen und effizienten Konsistenzprüfung bereits zur Entwurfszeit.

1) Zwei abstrakte Datenobjekte, die jeweils mittels implizitem ADT eingeführt werden, können somit nie typäquivalent sein: Zu jedem implizitem ADT existiert genau ein abstraktes Datenobjekt dieses Typs.

2) Es lehnt sich an das von ADA (/DoD80/) für Datentypen im Bereich des Programmierens-im-Kleinen an.

Definition 2.1

- a) Zwei abstrakte Datenobjekte *ADO1* und *ADO2*, die bzgl. eines explizit deklarierten abstrakten Datentyps *ADT* typäquivalent sind, heißen *ADT-äquivalent*.
- b) Sei *ADT* ein explizit deklariertes abstrakter Datentyp. Dann sei *adt_ADOM* die Menge aller *ADT*-äquivalenten abstrakten Datenobjekte.

Im weiteren Verlauf spielen auch abstrakte Datentypen eine Rolle, die Ausprägungen ein- und desselben GADT's (bei evtl. unterschiedlichen Aktualparametrisierungen) sind, also Ähnlichkeit in ihrer Struktur aufweisen. Dabei sollen die abstrakten Datentypen auch über mehrere Konkretisierungsstufen aus dem GADT gewonnen werden können: Zwei explizit deklarierte abstrakte Datentypen *ADT1* und *ADT2* heißen *strukturähnlich*, wenn sie das Resultat einer direkten oder indirekten Konkretisierung desselben GADT's darstellen.

Beispiel (*T_INT_STACK* und *T_REAL_STACK* sind strukturähnlich):

```
adt T_INT_STACK : GT_STACK (INTEGER);  
adt T_REAL_STACK : GT_STACK (REAL);
```

Die Strukturähnlichkeit überträgt sich auch auf Objekte:

Zwei abstrakte Datenobjekte *ADO1* und *ADO2* sollen dann *strukturähnlich* heißen, wenn es einen GADT *GADT* gibt, so daß *ADO1* und *ADO2* Objekte zweier abstrakter Datentypen sind, die bzgl. *GADT* strukturähnlich sind, oder direkt mittels *GADT* deklariert sind.¹⁾

Definition 2.2

- a) Zwei abstrakte Datenobjekte *ADO1* und *ADO2*, die bzgl. eines GADT's *GADT* strukturähnlich sind, heißen *GADT-strukturiert*.
- b) Sei *GADT* ein generischer abstrakter Datentyp. Dann sei *gadt_ADOM* die Menge aller *GADT*-strukturierten abstrakten Datenobjekte.

¹⁾ Die Strukturähnlichkeit basiert also (analog der Typäquivalenz) nicht auf textueller Ähnlichkeit. Typäquivalenz und Strukturähnlichkeit treten also nur auf, wenn sie der Entwickler als solche erkannt hat.

Beispiel:

Aus dem GADT *GT ASSMEM* seien durch verschiedene Konkretisierungen zwei unterschiedliche Ausprägungen *T ASSMEM1* und *T ASSMEM2* gewonnen worden. Zwei Assoziativspeicher-Objekte *ASSMEM1* vom ADT *T ASSMEM1* und *ASSMEM2* vom ADT *T ASSMEM2* sind dann strukturähnlich, nämlich *GT ASSMEM*-strukturiert. Durch *gt assmem* ADOM werden die Assoziativspeicher-Objekte aller abstrakten Datentypen zusammengefaßt, die Ausprägungen von *GT ASSMEM* sind.

2.2.2.4 DYNAMISCHE DATENSTRUKTUREN

Dynamische Datenstrukturen sind ebenfalls mit Hilfe der Methode der Datenabstraktion spezifizierbar.

Eine einzelne dynamische Datenstruktur kann als ADO aufgefaßt werden, dessen Repräsentierung jedoch erst zur Laufzeit des Systems (im Haldenspeicher) aufgebaut wird:

representation
dynamic

Die (vernetzten) Elemente der Datenstruktur sind i.a. wiederum (dynamisch erzeugte) abstrakte Datenobjekte bestimmter abstrakter Datentypen.¹⁾

Beispiel:

Der *ASSMEM*-Namenspeicher ordnet externen Entity-Namen interne zu. Er stellt ein ADO *NM* mit Operationen wie *INIT*, *INSERT ASSOCIATION*, *DELETE ASSOCIATION*, *DUMP* usw. dar. Eine mögliche Realisierung könnte auf dem Konzept des B-Baums /BACR72/, einem balancierten Vielwegbaum, beruhen (Abb. 2-17 zeigt die entsprechende Darstellung im Systemgraph). Die Knoten eines solchen B-Baums würden im vorliegenden Anwendungsfall einige externe Namen (Schlüssel), interne Namen (Schlüsseln zugeordnete Informationen) bzw. Verweise auf Folgeknoten enthalten und sind von einem abstrakten Datentyp *T NMNODE*. *T NMNODE* hat Operationen wie *INSERT_EXTNAM*, *CHANGE_INTNAM* oder *READ_SUCCESOR* für Zugriffe auf den Inhalt von

¹⁾ Jeder dieser abstrakten Datentypen hat in seinem Operationssatz Operationen zum Erzeugen und Löschen eines Objekts dieses ADT's vorzusehen.

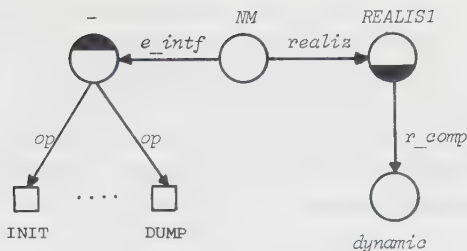


Abb. 2-17

Knoten(-objekten) vorzusehen. Da der *NM* realisierende B-Baum als dynamische Datenstruktur erst zur Laufzeit aufgebaut wird, muß *T_NMNODE* auch eine Operation *GENERATE NMNODE* zum Erzeugen und Initialisieren eines Knotenobjekts und eine Operation *FREE_NMNODE* zum Löschen eines Knotenobjekts bereitstellen.

Der Typ *T_NMNODE* wird sinnvollerweise in der inneren Schnittstelle von *NM* eingeführt, da er nur lokal in *NM* benutzt wird. Funktionen, die die Implementierung der *NM*-Operationen erleichtern und auf der B-Baum-Struktur operieren, wie *SPLIT NMNODE* (Splitten eines Knotens in zwei halbgefüllte), *MERGE NMNODES* (Verschmelzen zweier Knoten zu einem), sind ebenfalls Bestandteil der inneren Schnittstelle.

Die Notation der Spezifikation von *NM* ist mit der zur Spezifikation statischer Datenstrukturen identisch.

Natürlich könnte *NM* auch über einen entsprechenden expliziten ADT deklariert werden: Steht bereits in der Systembibliothek ein GADT *GT_B_TREE* (z.B. mit den formalen generischen Parametern *T_KEY*, *T_INFO* und *KEYNUMB*) zur Verfügung, ließe sich *NM* als Objekt des Typs *GT_B_TREE*, z.B. mit der Aktualparametrisierung (*T_EXTNAME*, *T_INTNAME*, 100), einführen.¹⁾ 2)

2.2.2.5 ABSTRAKTE FUNKTION

Eine *abstrakte Funktion* ist ein Algorithmus, der auf einer bestimmten Abstraktionsebene zur Konstruktion des Problemlösungs-Algorithmus beiträgt.

Sie kann auch als Befehl einer abstrakten Maschine gesehen werden,

1) Die Repräsentierung dieses GADT's ergibt sich natürlich auch erst dynamisch und ist von Objekt zu Objekt verschieden.

2) Die auf *NM* definierten Operationen sind dann natürlich nicht mehr Namenspeicher-spezifisch bezeichnet: *INSERT_KEYWITHINFO*, *DELETE_KEYWITHINFO*, *DUMP_B_TREE*.

der abstrakte Datenobjekte derselben abstrakten Maschine als *Argumente* besitzt und evtl. deren Zustände verändert. Sie hat insofern kein "Gedächtnis" (im Gegensatz zu abstrakten Datenobjekten), als ihr Effekt ausschließlich vom Momentan-Zustand ihrer Argumente abhängt.

Manche Argumente einer abstrakten Funktion können von Berechnung zu Berechnung stets dieselben sein.¹⁾

Beispiel:

Das Lösen einer Dateneinheit aus ihrer Vernetzung mit den übrigen Dateneinheiten in *ASSMEM* erfordert nicht wenige Aktionen. Da in mehreren *ASSMEM*-Operationen Dateneinheiten freizugeben sind, bietet es sich an, eine abstrakte Funktion *RELEASE_ENT* zu formulieren. Sie operiert stets auf den fünf Objekten der *ASSMEM*-Repräsentierung.

Andere Argumente können von Berechnung zu Berechnung wechseln, sind also aktuelle ADO-Parameter²⁾ der abstrakten Funktion. Aus algorithmisch-logischen und syntaktischen Gründen müssen dabei die abstrakten Datenobjekte, die bei allen Berechnungen aktuell auf der i-ten Parameterposition übergeben werden, (zwar nicht unbedingt typäquivalent, jedoch zumindest) strukturähnlich³⁾ sein.

Beispiel:

Wie gesehen, ist in der inneren Schnittstelle des GADT's *GT_B_TREE* eine abstrakte Funktion *MERGE_NODES* zum Verschmelzen zweier B-Baum-Knoten einzubringen. Diese Knoten sind Objekte eines ADT's, der als Ausprägung des GADT's *GT_B_TREE_NODE* gewonnen wird, wobei *GT_B_TREE_NODE* die formalen generischen Parameter *T_KEY*, *T_INFO* und *KEYNUMB* besitzt. Sinnvollerweise ist *MERGE_NODES* unabhängig von der Aktualparametrisierung von *GT_B_TREE_NODE* auf sämtlichen *GT_B_TREE_NODE*-strukturierten Objekten definiert.

Die Spezifikation einer abstrakten Funktion *FUNC* erfordert also (u.a.) die Festlegung ihres Definitionsbereichs *func_DEF*, wobei

-
- 1) Analog den globalen Objekten eines Unterprogramms im Bereich des Programmierens-im-Kleinen.
 - 2) Diese Parameter sind Laufzeit-Parameter.
 - 3) Diese Form der Spezifikation der zulässigen aktuellen ADO-Parameter wurde erstmals in der Programmiersprache ALPHARD (/WULO76/) vorgeschlagen. Bei /ALT78/ kommen all die abstrakten Datenobjekte als Aktualparameter einer abstrakten Funktion ("F-Modul") in Frage, die (evtl. rein zufällig) Operationen besitzen, die mit denen des entsprechenden Formalparameters gleichbezeichnet sind. Diese rein formal-syntaktische Bedingung erscheint als zu schwach.

$$\text{func_DEF} := \overbrace{(g)\text{adt1_ADOM} \times \dots \times (g)\text{adt_ADOM} \times \text{AD01} \times \dots \times \text{AD01}}^{1)}$$

Spezifikation der aktuellen Parameter (wechselnde Argumente) feste Argumente

Eine abstrakte Funktion kann zur Realisierung von abstrakten Datenobjekten, abstrakten Datentypen bzw. anderen Funktionen *benutzt* werden. Diese erhalten aber keinen Einblick in algorithmische Details der Funktion (Schutz des Algorithmus).

Zur Formulierung des Algorithmus können neben weiteren abstrakten Funktionen auch lokale abstrakte Datenobjekte notwendig sein, deren Gültigkeit sich auf diese Funktion beschränkt und nur bis zur Beendigung eines entsprechenden Funktionsaufrufs existieren.

2.2.2.6 (PARAMETRISIERTE) FUNKTIONSKLASSE

Logisch zusammengehörige abstrakte Funktionen lassen sich zu einer *Funktionsklasse* (FK) zusammenfassen.²⁾ Es ist schwierig, ein Kriterium für diese "Zusammengehörigkeits-Eigenschaft" anzugeben. Ein Indiz dafür könnte sein, daß die Funktionen in den Typen ihrer formalen Parameter/festen Argumente Gemeinsamkeiten aufweisen. Abstrakte Funktionen einer FK operieren also i.a. mindestens auf einer gemeinsamen Objektmenge, charakterisiert durch den Objekttyp.

Beispiele:

- In *ASSMEM* sind neben *RELEASE_ENT* noch weitere abstrakte Funktionen zur Unterstützung der Implementierung der *ASSMEM*-Operationen vorgesehen. Da auch sie auf den fünf Objekten der *ASSMEM*-Repräsentierung als festen Argumenten operieren (und auch ein logischer Bezug zwischen ihnen existiert), können sie als Funktionsklasse spezifiziert werden.
- In einem Grafiksystem zur interaktiven Erstellung von Grafik-Bildern können elementare Bilder zu komplexeren (dynamisch) zusammengesetzt werden. Die Informationen zu jedem Bild werden jeweils in einer (dynamischen) abstrakten Datenstruktur vom Typ *PICT* gehalten. Folgende Funktionen könnten Elemente einer Funktionsklasse *PICTSYN* zur Bildsynthese sein:

1) $(g)\text{ADTi}$ ($i \leq \{1, \dots, k\}$) sind (generische) abstrakte Datentypen.

2) /ALT78/ kennt keine Bündelung von "F-Moduln", so daß das vorgestellte Konzept in dieser Hinsicht allgemeiner ist.

```

1)
ROTATE (ado P1 : PICT by .. OPi, ..) /* Rotation eines Bildes */
COMPOSE (ado P1, P2 : PICT by .. OPj, ..;
        ado P3 : PICT by .. OPk, ..) /* Komponieren von Bildern */

```

Während die Funktionsklasse des ersten Beispiels stets auf fest fixierten abstrakten Datenobjekten operiert, ist die Funktionsklasse des zweiten Beispiels auf der Menge aller Objekte des ADT's *PICT* definiert: *PICTSYN* (set of *P* : *PICT*). An der Stelle der Benutzung der Funktionsklasse wird (als Aktualisierung von set of *P* : *PICT*) zu präzisieren sein, welche *PICT*-Objekte im einzelnen als aktuelle Laufzeit-Parameter für die Funktionen der Funktionsklasse in Frage kommen. Generell kann eine Funktionsklasse natürlich mehrere Formalparameter besitzen. Eine Funktionsklasse mit solchen Parametern, die die wechselnden Argumente ihrer Funktionen spezifizieren, heißt *parametrisierte Funktionsklasse* (kurz PFK).

Bemerkung: Die abstrakten Funktionen "Zuweisung" und "(Un-)Gleichheitstest" sind auf je zwei Objekten eines, beliebigen, aber festen ADT's implizit definiert. Sie brauchen nicht als Funktionen einer Funktionsklasse eingebracht werden.

Eine (parametrisierte) Funktionsklasse *FK* wird als Systemkomponente der Art *fk* bzw. *pfk* (i.a. in der inneren Schnittstelle einer anderen Systemkomponente) spezifiziert.

Die *FK*-Funktionen werden in der äußeren Schnittstelle von *FK* eingebracht:

```

provides
  func FUNC1 (.....
    ado F_ADOi : ADTi by .. OPij, ..
    ..... );
    .
    .
    .
  func FUNCn (.....
    ado F_ADonk : ADTk by .. OPk1, ..
    ..... );

```

1) Es sind die Operationen zu spezifizieren, die entsprechende Aktualparameter mindestens besitzen müssen.

Zur inneren Schnittstelle der Realisierung:

- Die spätere Implementierung der Funktionen von *FK* kann die Einführung lokaler Funktionsklassen *LOC_FKi* nahelegen ("Hilfsdienste"). Diese *LOC_FKi* werden in der inneren Schnittstelle eingebracht:

```
local functions  
fk  LOC_FKi;  
    ⋮  
fk  LOC_FKl;
```

Lokale abstrakte Funktionen, die aus logischen Gründen nur schwer einer Funktionsklasse zuordenbar sind (z.B. eine Fehlerbehandlungs-Funktion), können ebenfalls hier spezifiziert werden¹⁾:

```
func LOC_FUNC (...);
```

Neben (parametrisierten) Funktionsklassen können auch **abstrakte** Datenobjekte und (generische) **abstrakte** Datentypen lokale Funktionen/Funktionsklassen besitzen.

Abb. 2-18 zeigt die Darstellung lokaler Funktionen/Funktionsklassen im Systemgraph. Eine Systemkomponente der Art *fk* wird durch die Knoten-Grundform  repräsentiert. Die Kantenmarkierung *l_func* steht für "local function".

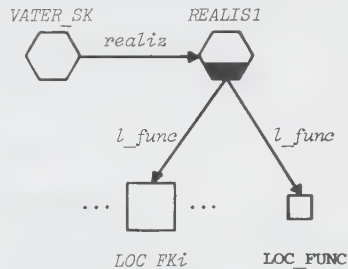


Abb. 2-18

¹⁾ Somit wird vermieden, solche Funktionen "künstlich" zu Funktionsklassen aufzublähen, die eben nur diese Funktion bereitstellen.

Die Implementierung der *FK*-Funktionen kann auch lokale abstrakte Datenobjekte *LOC_ADO_j* nötig machen, die nur bis zur Beendigung eines Funktionsaufrufs existieren. Sie werden mit Hilfe lokaler (generischer) abstrakter Datentypen *LOC_ADT_k* deklariert, die in der inneren Schnittstelle einzuführen sind:

```
local data types
  adt LOC_ADT1 : ...;
  :
  adt LOC_ADTl : ...;
```

Beispiel:

Die parametrisierte Funktionsklasse *PICTSYN* aus obigem Beispiel könnte wie in Abb. 2-19 spezifiziert werden.

```
pfk PICTSYN (set of P : PICT)

  external interface

    provides
      func ROTATE (ado P1 : PICT by ... OPi, ...);
      func COMPOSE (ado P1,P2 : PICT by ... OPj, ...;
                   ado P3 : PICT by ... OPk, ...);
      :
      :

  end external interface

  pfk-realization REALIS1

    internal interface

      local data types
        :
        adt LOC_ADTi : ...;
        :

      local functions
        :
        pfk LOC_PFKj : ...;
        :
        :

    end internal interface

  end pfk-realization

end pfk PICTSYN
```

Abb. 2-19: parametrisierte Funktionsklasse *PICTSYN*

Darstellung im Systemgraph:

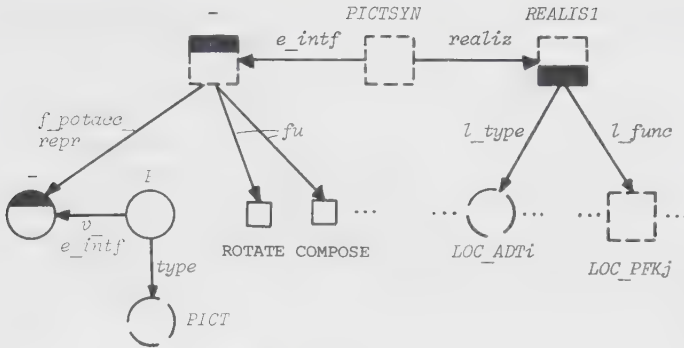


Abb. 2-20

Die Systemkomponenten-Art pfk wird im Systemgraph durch die Knotenform  ausgedrückt. Ansonsten entspricht die Darstellung

einer PFK-Deklaration der einer ADO-Deklaration mittels implizitem ADT (siehe Abschn. 2.2.2.1). Nur die Kanten, die in der äußeren Schnittstelle die Funktionen repräsentieren, sind mit *fu* (function) markiert.

Der Knoten für die äußere Schnittstelle von *PICTSYN* wird über eine *f_potace repr*-Kante mit dem Knoten für die äußere Schnittstelle von *P* verbunden (*P* repräsentiert die als Aktualparameter zu übergebenden abstrakten Datenobjekte). *P* selbst wird durch die bekannte Systemgraph-Teilstruktur für eine ADO-Deklaration mittels explizitem ADT abgelegt. Auf die Bedeutung der Markierung der *f_potace repr*-Kante wird später noch eingegangen. Dann wird auch deutlich, warum gerade die Knoten für die äußeren Schnittstellen verbunden werden.

Die formalen ADO-Parameter der Funktionen werden ähnlich berücksichtigt. Abb. 2-21 zeigt beispielsweise den ADO-Parameter *P1* der Funktion *ROTATE*. Die Bedeutung der Markierung der *f_potace*-Kante wird ebenfalls erst später klar.

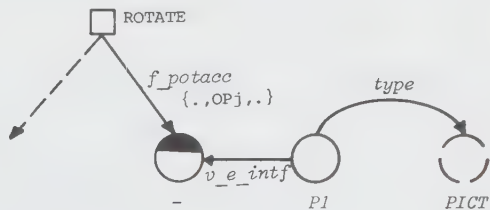


Abb. 2-21

Die Parameter einer PFK erwiesen sich als Platzhalter für abstrakte Datenobjekte, die bei einer Benutzung der PFK als potentielle Aktualparameter der PFK-Funktionen ausgewählt werden. Die PFK-Parameter dienen also der Spezifikation von Laufzeit-Parametern.

Im Gegensatz dazu könnte man sich - analog generischen abstrakten Datentypen - Funktionsklassen vorstellen, deren Parameter generisch, also Entwurfszeit-Parameter sind. Werden bei der Benutzung einer solchen FK diese Parameter dann aktualisiert, bedeutet dies, daß zunächst offengelassenen Größen (z.B. Zugriffe auf unbestimmte abstrakte Datenobjekte) noch zur Entwurfszeit endgültig fixiert werden. Auf solche *generische Funktionsklassen* wird nicht weiter eingegangen.

ERGÄNZENDE BEMERKUNGEN

- 1) Zur Weitergabe logisch zusammengehöriger einfacher Datenobjekte und -typen könnte man als weitere Art von Systemkomponenten ein *Daten- und Typenpaket* ähnlich eines erweiterten FORTRAN-COMMON's vorsehen.
- 2) Die *Initialisierung* von abstrakten Datenobjekten kann bereits in der Spezifikation in Form eigener Initialisierungs-Operationen berücksichtigt werden. Alternativ dazu könnten in der Implementierungsphase die ADO-Deklarationen um die Zuweisung von Anfangswerten ergänzt werden.

2.2.3 BEZIEHUNGEN ZWISCHEN SYSTEMKOMPONENTEN

Nach der Charakterisierung der Arten an Systemkomponenten sind nun die Beziehungen der Systemkomponenten untereinander zu erörtern.

2.2.3.1 SYSTEMHIERARCHIE

Zwischen zwei Systemkomponenten $SK1$ und $SK2$ besteht eine *Realisiert-* (oder *Definiert-*) *Beziehung* (*R-Beziehung*), falls $SK1$ mit Hilfe von $SK2$ realisiert (oder definiert) wird bzw. falls $SK2$ zur Realisierung (oder Definition) von $SK1$ beiträgt ($SK2$ ist also lokale Ressource von $SK1$).

In Abschnitt 2.1.2 wurde für "wohlstrukturierte" Softwaresysteme eine (strenge) Hierarchie bzgl. der Realisiert-Beziehung gefordert. Diese Hierarchie ergibt sich aus der internen Struktur der Systemkomponenten, was anhand des Modells eines hierarchischen Graphen /PRA69/ verdeutlicht werde.

Es sei kurz die Idee dieser speziellen Graphenart erläutert. Manche Sachverhalte legen eine Darstellung in Form eines Graphen nahe, wobei die Knoten dieses Graphen wiederum Graphen als Inhalt enthalten.

Als Beispiel sei die Struktur eines Programms in einer blockstrukturierten höheren Programmiersprache genannt. Der Kontrollfluß des Hauptprogramms und sämtlicher Unterprogramme werde jeweils als eigener (Programmablauf-)Graph dargestellt. Ein Interpretierer führt nun dieses Programm aus, indem er die Anweisungen abarbeitet, die durch den Inhalt der einzelnen Knoten repräsentiert werden. Die Abarbeitung eines Unterprogramm-Aufrufs löst dabei die Abarbeitung des zugehörigen Unterprogramm-Rumpfes aus. Dies legt jedoch die Vorstellung nahe, daß der entsprechende Aufruf-Knoten eigentlich einen ganzen Graphen, nämlich den des Unterprogramm-Rumpfes, als Inhalt besitzt.

Ein *hierarchischer Graph* (*H-Graph*) ist also ein knoten-/kantenmarkierter, gerichteter Graph, dessen Knoteninhalte entweder elementar oder wieder Graphen sind.

Die Hierarchie der Systemkomponenten in einem Softwaresystem läßt sich direkt als H-Graph veranschaulichen: Jedes abstrakte Datenobjekt ADO und jede (parametrisierte) Funktionsklasse (P) FK der abstrakten Maschine AM_1 werden durch Objekte und Funktionsklassen der abstrakten Maschine AM_{i-1} realisiert. Die Systemgraph-Ausschnitte, die diesen Sachverhalt beschreiben, könnten als Inhalt entsprechen-

ADO

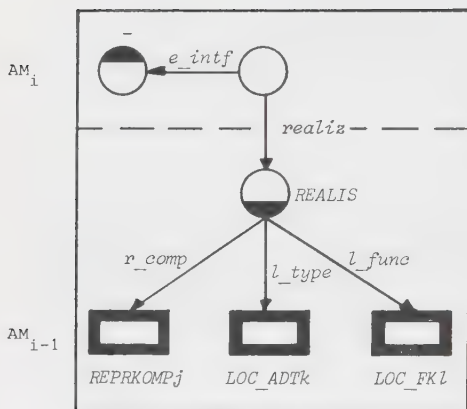


Abb. 2-22a

SYSTEM

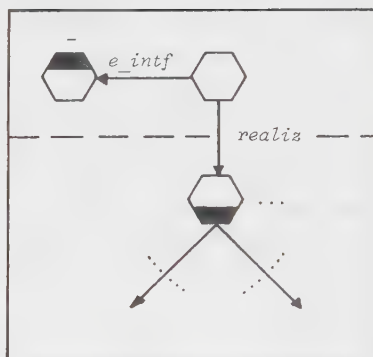


Abb. 2-22b

der H-Graph-Knoten repräsentiert werden. Abb. 2-22a zeigt beispielsweise den H-Graph-Knoten für $ADO^{1)}$. Die gesamte Systemhierarchie wird dann durch einen H-Graph mit solchen Knoten und dem Knoten für das Gesamtsystem $SYSTEM$ (Abb. 2-22b) aufgespannt. Die Knoten für die Objekte, Typen und Funktionen der abstrakten Maschine AM_0 , der Implementierungssprache also, bilden die unterste Hierarchieebene der H-Graph-Knoten.

Die *Streckung* eines H-Graph-Knotens kn im Inhaltsgraphen eines anderen H-Graph-Knotens bedeutet, anstelle kn dessen Inhalt einzukopieren. Besteht dieser Inhalt wiederum aus einem Graph, wird der Ersetzungsprozeß iterativ für dessen Knoten und deren Inhaltsgraphen fortgesetzt solange, bis nur mehr Knoten mit elementarem Inhalt zu ersetzen sind.

Der H-Graph-Knoten kn repräsentiere die Systemkomponente SK . Der Systemgraph-Ausschnitt, den man durch Streckung von kn erhält, beschreibt dann die Hierarchie des durch SK aufgespannten Teilsystems (*Teilhierarchie*).

Die Streckung des Knotens der Systemkomponente $SYSTEM$ bewirkt das

¹⁾ H-Graph-Knoten mit einem Graph als Inhalt werden durch  gekennzeichnet.

Aufspannen der Hierarchie des *Gesamtsystems* (*Systemhierarchie*). Das Ergebnis dieser Streckung ist identisch mit der Systemgraph-Darstellung von *SYSTEM*.

Wie in Abschnitt 2.2.2.2 angedeutet, kann ein lokales ADO *ADO* mittels eines expliziten ADT's *ADT* eingeführt werden (Abb. 2-4).

SK

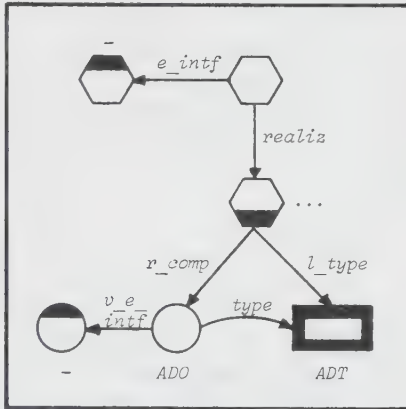


Abb. 2-23

Der H-Graph-Knoten zu *ADO* ist in diesem Fall bereits elementar: *ADO* besitzt keine eigene Realisierung. Durch *ADO* wird somit keine neue Ebene in der Realisiert-Hierarchie eingeführt (wohl jedoch durch *ADT*).

Beispiel:

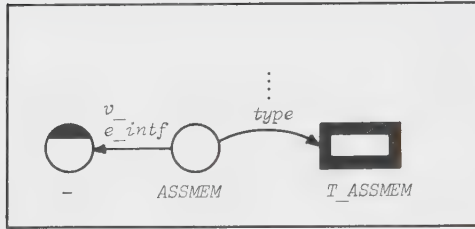
In Abschnitt 2.2.2.4 wurde der Namenspeicher *NM* des Assoziativspeichers *ASSMEM* als B-Baum realisiert. In /ENW74/, /GAL78/ wird eine alternative Realisierung als Tabelle mit Hashzugriff angegeben. Dabei ist diese Tabelle als Feld¹⁾ einer Anzahl gleichstrukturierter Teiltabellen (Namenspeicher-Seiten) *NMPAGE_i* vom ADT *T_NMPAGE* aufgebaut. *T_NMPAGE* kann seinerseits noch weiter verfeinert werden.

UTIL stellt auf der *NM*-Repräsentierung operierende Hilfsdienste bereit (z.B. die Hashfunktion zur Adressierung der Tabelleneinträge).

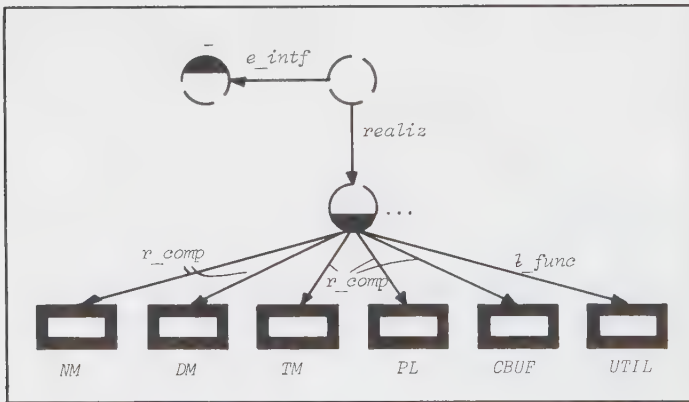
Abb. 2-24 zeigt den entsprechenden H-Graph für den ADT *T_ASSMEM*.

¹⁾ Dadurch sei nur die Strukturierungsform, nicht ein Implementierungsaspekt angesprochen.

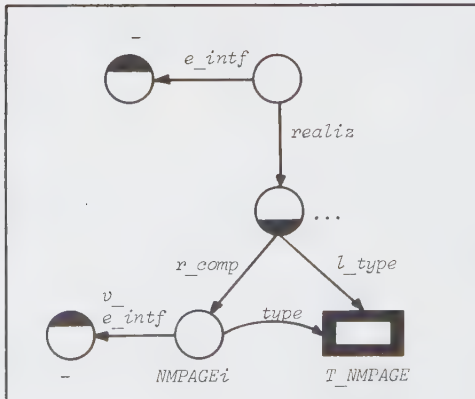
Anwendungs-System



T_ASSMEM



NM



In Anlehnung an /ALT78/ nennen wir ein Softwaresystem mit einem ADO bzw. (G)ADT als Systemwurzel *datenorientiert*, mit einer (P)FK an der Systemspitze *funktionsorientiert*.

Beispiele für ein datenorientiertes System sind ein B-Baum (-objekt/-typ) bzw. der schon öfters angeführte Software-Assoziativspeicher, während ein Compiler ein funktionsorientiertes System darstellt: Der Compilerrahmen (Hauptprogramm) wird als Funktion *COMPILER* einer Funktionsklasse spezifiziert, die ein Programmiersystem realisiert und noch weitere Funktionen wie *EDITOR*, *BINDER* und *LADER* bereithält. *COMPILER* führt lokal Systemkomponenten für die verschiedenen Compilerlisten, die Syntaxanalyse- und die Codeerzeugungs-Prozeduren ein.

2.2.3.2 ZUGRIFFSRECHT-BEZIEHUNG

Der hier vorgestellte Software-Arbeitsplatz soll die Spezifikation sicherer Softwaresysteme unterstützen, wobei mit "sicher" in erster Linie Datensicherheit zu verstehen ist. Daten in unserem Strukturmodell sind aber die abstrakten Datenobjekte. Diese "Informationsbehälter" sind durch Zugriffe zur Laufzeit manipulierbar. Es gilt also zu klären, welche Bedingungen hinsichtlich aller dynamisch möglichen Zugriffe, also welches *Zugriffsmodell*, ein Softwaresystem zu einem *zugriffssicheren* System machen.

Ob alle dynamischen Zugriffe im zu entwickelnden System mit dem Zugriffsmodell konsistent sind, sollte bereits statisch, also zur Entwurfs- bzw. Implementierungszeit überprüft werden können. Somit könnte eine Umstrukturierung des Systems, die aufgrund nicht intendierter Zugriffe zwischen Systemkomponenten evtl. notwendig wird, zu einem möglichst frühen Entwicklungszeitpunkt erfolgen.

Es gilt nun, das Zugriffsmodell zu präzisieren.

Besteht zwischen einer Systemkomponente *SK* und einem ADO *ADO* die Realisiert-Beziehung, so wurde *ADO* als lokale Ressource von *SK* eingeführt, da sie zur Implementierung von *SK* benötigt wird. Deshalb muß natürlich *SK* auch auf *ADO* zugreifen dürfen.

Die Praxis zeigt jedoch, daß eine Zugriffsstruktur, die allein durch die R-Beziehung impliziert wird, zu eingeschränkt ist. Für eine Sy-

Systemkomponente müssen außer ihren lokalen abstrakten Datenobjekten auch andere abstrakte Datenobjekte einer bestimmten Nachbarschaft (dynamisch) *zugreifbar* sein.

Beispielsweise dient *CBUF* im Softwaresystem *ASSMEM* als Puffer zur Kommunikation zwischen den Komponenten *NM*, *DM* und *TM*, muß also für die Bruderkomponenten¹⁾ zugreifbar sein. Strenggenommen müssen sogar die lokalen Komponenten von *NM*, *DM* und *TM* (bzw. die entsprechenden abstrakten Datentypen) auf *CBUF* zugreifen dürfen, ein Zugriff in der R-Hierarchie aufwärts also.

Welche sinnvolle Erweiterung der Zugriffsstruktur bietet sich an? Zunächst gilt es, das Prinzip des information hidings zu wahren: Eine Systemkomponente darf nicht auf die Interna seiner Sohn- bzw. seiner Bruderkomponenten zugreifen.

Einen ersten Ansatz für ein Zugriffsmodell, das dieses Prinzip berücksichtigt, liefert die Sichtbarkeitsregelung in blockstrukturierteren höheren Programmiersprachen: Für ein Unterprogramm sind (abgesehen von Verdeckungen aufgrund von Gleichbezeichnung) sämtliche Unterprogramme sichtbar, die (bzgl. der Enthaltenseins-Beziehung) Söhne, Brüder, direkte Vorfahren und Brüder dieser direkten Vorfahren sind. Außerdem ist ein Unterprogramm für sich selbst sichtbar (Rekursion möglich).

Wird jedoch diese Sichtbarkeitsregelung direkt als Regelung der Zugriffe in unserem Strukturmodell übernommen, fällt sofort auf: Die aufgrund der R-Hierarchie implizierte Struktur von Zugriffen ist zu wenig restriktiv. Während nach unten in der R-Hierarchie nur auf die direkten Söhne zugegriffen werden darf, ist der Zugriff in höhere Hierarchieebenen viel zu allgemein.²⁾ Das Entwurfsprinzip der kontrollierten Informationsverteilung, das bei der Spezifikation "wohlstrukturierter" Softwaresysteme einzuhalten ist (vgl. Abschnitt 2.1.3), läßt sich auf diese Weise nicht realisieren: Der Systementwickler besitzt keine Möglichkeit, die für eine Systemkomponente zugreifbaren abstrakten Datenobjekte gezielt einzuschränken, um exakt die intendierte Systemstruktur und nicht einen überflüssig hohen Grad an Zugriffs-Vernetzung zu erzielen.

Auch das Prinzip der Zugriffstransparenz ist nicht erfüllt. Dieses

1) "Bruder" bzgl. der R-Beziehung.

2) /NAG82/ zeigt auf, welches Chaos an Zugriffsmöglichkeiten sich bereits bei einem kleinen Beispiel ergeben würde.

Prinzip fordert, daß sämtliche Zugriffe einer Systemkomponente *SK* und aller Systemkomponenten des durch *SK* aufgespannten Teilsystems bereits vollständig aus der äußeren Schnittstelle von *SK* ersichtlich sind.

Die geschilderten Schwächen einer durch die R-Hierarchie gegebenen impliziten Zugriffsregelung legen es nahe, eine Zugriffskontrolle aufgrund einer expliziten Vergabe von Zugriffsrechten einzuführen (/ALT78/). Jeder Systemkomponente *SK* darf ein *Zugriffsrecht* (*ZR*) bzgl. eines ADO's *ADO* eingeräumt werden, falls *ADO* im (noch zu definierenden) Zugriffsbereich von *SK* liegt. Zwischen *SK* (dem *ZR-Subjekt*) und *ADO* (dem *ZR-Objekt*) besteht dann eine *Zugriffsrecht-Beziehung* (potentieller Zugriff). Um die Zugriffskontrolle selektiver gestalten zu können¹⁾, kann das Zugriffsrecht durch sog. *ZR-Attribute* auf bestimmte Operationen des *ZR-Objekts* beschränkt werden.

Beispielsweise könnte einem ADT *ADT* (u.a.) Zugriffsrecht bzgl. der abstrakten Datenobjekte *ADO1* und *ADO2*, und zwar jeweils nur Zugriffe über die Operationen *OP2*, *OP4* und *OP5* zugestanden werden²⁾:

```
adu  ADT
  external interface
    provides
      :
      :
    access rights
      :
      ado  ADO1, ADO2 by OP2, OP4, OP5;
      :
  end external interface
  :
```

Abb. 2-25

-
- 1) z.B. Spezifikation ausschließlich lesenden Zugriffs bzw. begrenzter Schreib-Erlaubnis.
- 2) Zugriffsrechte werden in der äußeren Schnittstelle des *ZR-Subjekts* spezifiziert.

In jedem Fall (also auch beim Fehlen von ZR-Attributen) kann das ZR-Objekt in Wertzuweisungen und (Un-)Gleichheitstests auftreten.¹⁾

Um nur schwer zu überblickende Seiteneffekte auszuschließen, ist die ZR-Beziehung nicht-transitiv: Zugriffsrechte, die in obigem Beispiel *ADO1* oder *ADO2* selbst besitzen, werden nicht automatisch *SK* eingeräumt.

Zugriffsrecht bedeutet im übrigen natürlich nicht Zugriffspflicht, sondern stellt nur eine notwendige Bedingung für einen *aktuellen Zugriff* dar. Ob also in der Implementation der Operationen/Funktionen einer Systemkomponente von deren Zugriffsrechten auch Gebrauch gemacht wird, indem entsprechende Zugriffsoperationen aufgerufen werden, ist in diesem Zusammenhang bedeutungslos.

Ein aktueller Zugriff besitzt - nebenbei bemerkt - insofern nur statischen Charakter, als er den Aufruf einer Zugriffsoperation im Quelltext der Systemimplementation bedeutet. Ob dieser dynamisch mögliche Zugriff zur Laufzeit auch zu einem *faktischen Zugriff* führt, ist zur Implementierungszeit nicht entscheidbar.

Welche abstrakten Datenobjekte sind nun für eine Systemkomponente mögliche ZR-Objekte? Lassen wir alle die abstrakten Datenobjekte als ZR-Objekte zu, die aufgrund der impliziten Zugriffsstruktur, die durch die Realisiert-Hierarchie gegeben ist, zugreifbar sind, ist zwar das Prinzip des information hidings, noch nicht jedoch das Prinzip der Zugriffstransparenz gewahrt. Die Sohn-Systemkomponente *SK'* einer Systemkomponente *SK* kann nämlich bzgl. eines ADO's, das in der R-Hierarchie mindestens so hoch wie *SK* angesiedelt ist, ein Zugriffsrecht besitzen, obwohl *SK* dieses nicht innehat. *SK'* kann auch mehr ZR-Attribute als *SK* erhalten haben, wenn *SK* und *SK'* Zugriffsrecht bzgl. des ADO's besitzen. In beiden Fällen sind mögliche Zugriffe aus dem Teilsystem heraus, das durch *SK* aufgespannt wird, nicht bereits in der äußeren Schnittstelle von *SK* erkennbar. Diese Schwierigkeit wird behoben, wenn der Sohn-Systemkomponente *SK'* Zugriffsrechte bzgl. hierarchiemäßig höher angesiedelten abstrakten Datenobjekten nur dann eingeräumt werden dürfen, wenn auch die Vater-Systemkomponente *SK* diese Zugriffsrechte (mit mindestens

¹⁾ Eine ausschließliche Beschränkung des Zugriffsrechts auf die ZR-Attribute ist denkbar, wird jedoch nicht weiter verfolgt.

denselben ZR-Attributen) besitzt (*Vererbung* von Zugriffsrechten).

Nun setzt sich der *Zugriffsbereich* einer Systemkomponente $SK^{1)}$, also die Menge der von ihr dynamisch zugreifbaren abstrakten Datenobjekte (und damit die Menge ihrer potentiellen ZR-Objekte) zusammen aus

- a) den ADO-Söhnen von SK
- b) den ADO-Brüdern von SK (aber nicht deren Nachkommen)
- c) den abstrakten Datenobjekten, die ZR-Objekte des Vaters von SK sind (mit Ausnahme von SK selbst).

Zur Ausnahme in c): (Direkt oder indirekt) rekursive Zugriffe eines ADO's auf sich selbst sind nicht sinnvoll. Es gibt im Bereich des Programmierens-im-Großen keine praktisch relevanten Fälle, in denen eine Operation OP_i eines ADO's von einer Operation OP_j (auch $i=j$) desselben ADO's ⁱ (in)direkt Gebrauch machen müßte. ^j Aufgrund der Vererbungsregel werden durch diese Ausnahme auch (unerwünschte) Zugriffe auf direkte Vorfahren in der Realisiert-Hierarchie ausgeschlossen.

Aus dem gleichen Grund sind Zugriffs-Zyklen zwischen ADO-Brüdern auszuschließen: Ein ADO $ADO1$ sollte nicht mit Hilfe eines ADO-Bruders $ADO2$ realisiert werden, der selbst (direkt oder indirekt) auf $ADO1$ zugreift. Die Zugriffsrecht-Beziehung ist also zyklensfrei.

Es läßt sich auch (als der "inverse" Zugriffsbereich) der Systembereich angeben, in dem Systemkomponenten ein Zugriffsrecht bzgl. eines bestimmten ADO's ADO eingeräumt werden kann (*Zugreifbarkeitsbereich* von ADO). Die Menge der potentiellen ZR-Subjekte von ADO besteht aus

- a) dem Vater von ADO .
- b) den Brüdern von ADO und (bei entsprechender Zugriffsrecht-Vererbung) deren Nachfahren.

Einer Systemkomponente kann ein Zugriffsrecht bzgl. jedes ADO's seines Zugriffsbereiches eingeräumt werden (eine Zugriffskontrolle aufgrund einer Zugriffsrecht-Vergabe schränkt also die Menge der potentiellen ZR-Objekte nicht ein). Es sind jedoch verschiedene Möglichkeiten zu unterscheiden, wie eine Systemkomponente SK ein Zugriffsrecht bzgl. eines ADO's ADO zugestanden bekommen kann.

¹⁾ Der Fall, daß SK Systemwurzel ist, wird später behandelt.

Führen Systemkomponenten lokale abstrakte Datenobjekte als Ressourcen zu ihrer Implementierung ein, benötigen sie natürlich auch ein entsprechendes Zugriffsrecht. Zur Verringerung des Schreibaufwands und zur Erhöhung der Übersichtlichkeit wird diesen Systemkomponenten bzgl. sämtlicher ADO-Söhne automatisch ein Zugriffsrecht (mit vollem Operationensatz) übertragen (*implizites Zugriffsrecht*). Bzgl. allen anderen abstrakten Datenobjekten im Zugriffsbereich kann (wie im Beispiel von Abb. 2-25) ein Zugriffsrecht nur explizit zugestanden werden (*explizites Zugriffsrecht*). Funktionsklassen können nur explizite Zugriffsrechte erhalten¹⁾ (bei parametrisierten Funktionsklassen existiert neben dieser Möglichkeit aufgrund ihrer Parametrisierung noch eine weitere, auf die später eingegangen wird).

Bleiben noch die Systemkomponenten, die mittels eines expliziten (G)ADT's deklariert werden, beispielsweise ein ADO *ADO* vom expliziten ADT *ADT: ADO* werden (aufgrund des Schablonen-Charakters von *ADT*) implizit sämtliche expliziten Zugriffsrechte übertragen, die *ADT* besitzt. Diese übertragenen Zugriffsrechte sind der virtuellen äußeren Schnittstelle von *ADO* zuzuordnen und heißen deshalb *virtuelle Zugriffsrechte*. Da die äußere Schnittstelle von *ADO* nur virtuell ist, lassen sich *ADO* keine (weiteren) expliziten Zugriffsrechte zuordnen.²⁾

Falls *ADT*³⁾ mittels eines GADT's *GADT* deklariert ist und deshalb selbst (nur) virtuelle Zugriffsrechte besitzt, werden diese an *ADO* weitergereicht. Im Gegensatz zum Herstellen einer Zugriffsrecht-Beziehung pflanzen sich also beim Herstellen einer ADO-Typ- bzw. Typ-Typ-Beziehung die Zugriffsrechte des Objekts der Beziehung zum Subjekt der Beziehung fort.

Es sei noch einmal betont: Alle drei Zugriffsrecht-Arten besitzen denselben Effekt, sie unterscheiden sich nur in der Form ihrer Vergabe.

1) Nur einzelne Funktionen einer FK besitzen auch implizites Zugriffsrecht bzgl. ihrer lokalen abstrakten Datenobjekte.

2) *ADO* besitzt natürlich implizite Zugriffsrechte bzgl. der Komponenten seiner Repräsentierungs-Instanz. Diese impliziten Zugriffsrechte wollen wir jedoch nicht weiter betrachten.

3) Für einen GADT gilt Entsprechendes.

Explizite Zugriffsrechte¹⁾ werden im Systemgraph dargestellt als (vgl. Beispiel in Abb. 2-25)²⁾:

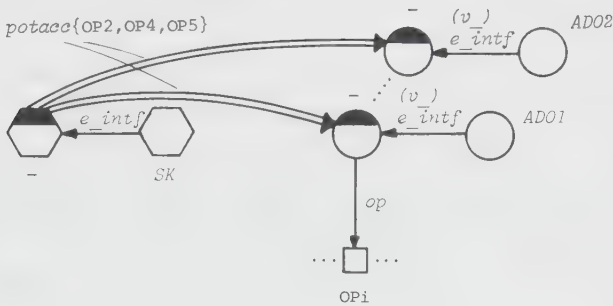


Abb. 2-26

Die Tatsache, daß Zugriffsrecht-Beziehungen zwischen zwei Systemkomponenten vollständig in deren äußeren Schnittstellen spezifiziert sind, spiegelt sich im Systemgraph wider: Ein explizites Zugriffsrecht wird durch eine *potacc*-Kante (*potential access*) zwischen dem Knoten für die äußere Schnittstelle des ZR-Subjekts und dem entsprechenden Knoten³⁾ des ZR-Objekts ausgedrückt. Die Markierung der *potacc*-Kante führt zusätzlich die ZR-Attribute als Markierungs-Attribute auf. Virtuelle Zugriffsrechte werden im Systemgraph wie explizite Zugriffsrechte (jedoch mit Kennenmarkierung *v_potacc* (*virtual potential access*)) dargestellt:

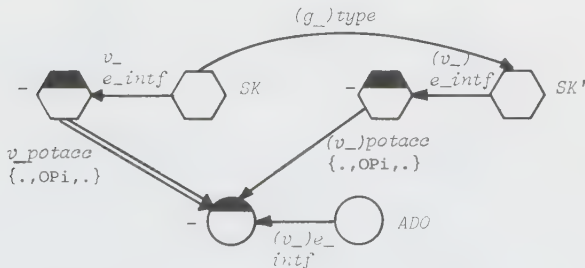


Abb. 2-27

¹⁾ Implizite Zugriffsrechte werden im Systemgraph nicht berücksichtigt.

2) Eine verstärkt gezeichnete Kante betone den jeweils interessierenden Sachverhalt.

3) Ist das ZR-Objekt mittels eines (G)ADT's deklariert, ist der Knoten für seine virtuelle äußere Schnittstelle Zielknoten.

Sämtliche expliziten bzw. virtuellen Zugriffsrechte des Objekts SK' einer ADO-Typ-Beziehung (*type*-Kante) bzw. einer Typ-Typ-Beziehung (*g_type*-Kante) "wandern" beim Herstellen der Beziehung über die (*g_type*-Kante (entgegen der Pfeilrichtung) zum Subjekt SK der Beziehung.

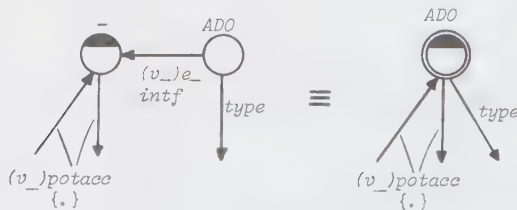
Wir wollen nun das Zugriffsmodell für "wohlstrukturierte" Softwaresysteme auch graphisch veranschaulichen.

In Abschnitt 2.2.3.1 (Systemhierarchie) wurde die hierarchische Realisiert-Struktur mit Hilfe des H-Graph-Modells beschrieben. Möchte man nun gleichzeitig Zugriffsrecht-Beziehungen mitberücksichtigen, erweist sich der klassische H-Graph als nicht ganz adäquates Beschreibungsmittel: Bezüge zwischen den Inhaltsgraphen verschiedener H-Graph-Knoten sind unzulässig (die Darstellung von Zugriffsrecht-Beziehungen zwischen Systemkomponenten verschiedener Hierarchieebenen beispielsweise sind derartige Bezüge). Genau diese Art von Bezügen erlaubt jedoch ein verallgemeinertes H-Graph-Modell, das Modell des *quasi-H-Graphen* (/GÖT77/).¹⁾

In Abb. 2-28 wird der Zugriffsbereich einer datenorientierten Systemkomponente SK für den Fall illustriert, daß die Struktur des zu entwickelnden Softwaresystems in Form eines quasi-H-Graphen repräsentiert wird.²⁾ Dabei beschreiben die $\Rightarrow$ -Kanten mögliche explizite Zugriffsrechte, die $\Rightarrow$ -Kante ein implizites Zugriffsrecht. Virtuelle Zugriffsrechte treten in der Abbildung nicht auf, da sämtliche Systemkomponenten mittels implizitem Typ bzw. Typdefinition deklariert worden seien.

Aus Übersichtlichkeitsgründen werden (wie auch in den folgenden Abschnitten öfters) die Knoten für die (virtuelle) äußere Schnittstelle einer Systemkomponente und für die Systemkomponente selbst als ein Knoten gezeichnet. Dieser Knoten übernimmt die Kanten, für die die Ausgangsknoten Quelle bzw. Ziel waren.

Beispiel:



1) Dies ist auch der einzige Unterschied zwischen beiden Modellen.

2) Der Übersichtlichkeit wegen wurde den Knoten des quasi-H-Graphen direkt ihr Inhaltsgraph einkopiert.

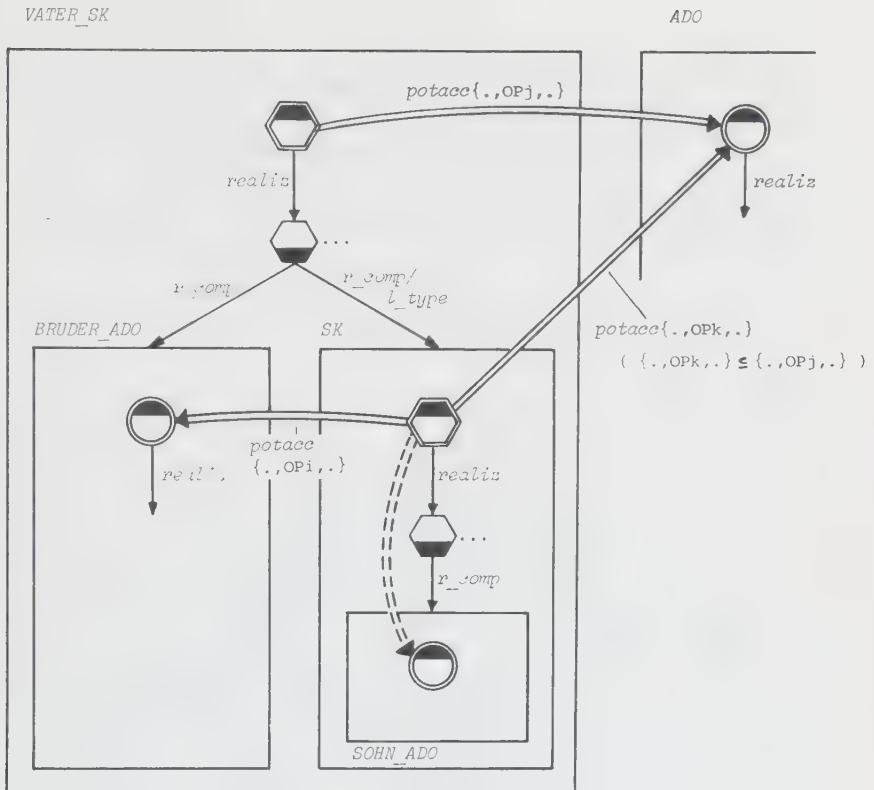


Abb. 2-28: Zugriffsbereich einer Systemkomponente

Beispiel:

Wir wollen einige der Zugriffsrecht-Beziehungen in unserem Assoziativspeicher *ASSMEM* studieren (Abb. 2-29). Die Suche nach einem internen Namen, der einem vorgegebenen externen Namen zugeordnet ist, läuft durch die *NM*-Operation *FIND ASSOCIATION* folgendermaßen ab: Der externe Name (Schlüssel) ist im Kommunikationspuffer *CBUF*, der dem Informationsaustausch zwischen *NM*, *DM* und *TM* dient, abgelegt. Eine in *UTIL* bereitgestellte Hashfunktion ermittelt aus dem externen Namen die Namenspeicherseite *NMPAGEi*, die die Zuordnung externer Name $\rightarrow$ interner Name enthält. Die ermittelte Seite wird unter Zugriff auf die Pagetabelle *PL* aktiviert. Eine Operation von *NMPAGEi* überträgt den gefundenen internen Namen nach *CBUF*.

Abb. 2-29: Zugriffsrecht-Beziehungen in ASSMEM

Der Namenspeicher *NM* benötigt also ein (explizites) Zugriffsrecht bzgl. der Pagetabelle *PL* (Seitenaktivierung) und bzgl. *CBUF*. Zwar machen die *NM*-Operationen von letzterem Zugriffsrecht nicht selbst Gebrauch, wohl jedoch die Operationen des Typs einer Namenspeicherseite *T NMPAGE* bzw. die Funktionen von *UTIL*. Wegen des Prinzips der Zugriffstransparenz (Vererbungsregel) ist deshalb auch *NM* dieses Zugriffsrecht einzuräumen. Das explizite Zugriffsrecht von *T NMPAGE* bzgl. *CBUF* wird jeder Namenspeicherseite *NMPAGE_i* als virtuelles Zugriffsrecht übertragen ($\Rightarrow$ -Kante).

Der Systemgraph ist die Darstellung eines Softwaresystems, die aus der Streckung des entsprechenden quasi-H-Graphen als äquivalente Beschreibungsform resultiert.

Im Teilsystem, das von einer Systemkomponente *SK* aufgespannt wird, sind Zugriffe auf abstrakte Datenobjekte außerhalb des Teilsystems nur möglich, falls *SK* entsprechende Zugriffsrechte besitzt. Vor allem für die Testphase ist eine noch größere Zugriffstransparenz erwünscht: Es interessiert, auf welche Systemkomponenten, die zur Realisierung von *SK* beitragen, sich die einzelnen Zugriffsrechte von *SK* weitervererben, also das Netz aller dynamisch möglichen Zugriffe innerhalb der Realisierung von *SK* (vgl. auch /ALT78/). Dieses Netz sollte bereits zur Entwurfszeit ermittelbar sein. Zugriffsrechte, die den lokalen Systemkomponenten von *SK* bzw. von *SK* zugegriffenen oder benutzten Systemkomponenten eingeräumt sind, sind den (virtuellen) äußeren Schnittstellen dieser Systemkomponenten entnehmbar. Unspezifiziert ist aber noch, auf welchen der *ZR*-Objekte von *SK* lokale bzw. von außerhalb benutzte abstrakte Funktionen einer PFK (außer ihren festen Argumenten) zur Laufzeit operieren können sollen. Dies wird in einer sog. *Benutzungsspezifikation* präzisiert.

In der Spezifikation einer PFK stehen Formalparameter für die abstrakten Datenobjekte, die als Aktualparameter der PFK-Funktionen auftreten können (siehe Abschn. 2.2.2.6). Bevor nun Funktionen dieser PFK benutzt werden dürfen, ist in der Benutzungsspezifikation für jeden dieser Formalparameter zu spezifizieren, welche abstrakten Datenobjekte (vom Typ des Formalparameters) Kandidaten für die Aktualparametrisierung der PFK-Funktionen zur Laufzeit sind. In einer Benutzungsspezifikation werden also der zu benutzenden PFK Zugriffsrechte in Form *aktueller ZR-Parameter* für die entspre-

chenden *formalen ZR-Parameter* übertragen.^{1), 2)} Dies könnte im Falle der PFK *PICTSTN* aus Abb. 2-19 folgendermaßen aussehen:

```
internal interface
:
:
  use specifications
:
  PICTSYN (PICTURE1, PICTURE2 by ..., OP1, ...;
          PICTURE3 by ..., OPm, ...);
:
end internal interface
```

Abb. 2-30

Es werden in der inneren Schnittstelle einer Systemkomponente *SK* den Funktionen der PFK *PICTSYN*³⁾ Zugriffsrechte bzgl. der *PICT*-Objekte *PICTURE1* und *PICTURE2* mit den ZR-Attributen {..., OP1, ..} bzw. bzgl. *PICTURE3* mit den ZR-Attributen {..., OPm, ..} eingeräumt.

Das Ergebnis der Aktualparametrisierung einer PFK ist eine (unbenannte) FK, da die Menge sämtlicher ihrer ZR-Objekte (Argumente) nun festliegt.⁴⁾

Jeder formale ZR-Parameter⁵⁾ einer einzelnen PFK-Funktion wird bei deren Aufruf durch ein aktuelles Zugriffsrecht aus der Menge von Zugriffsrechten abgesättigt, die bei der entsprechenden Benutzungsspezifikation spezifiziert wurde. Natürlich müssen dabei die ZR-Attribute des aktuellen Zugriffsrechts die des entsprechenden formalen Zugriffsrechts überdecken. Die aktuellen Zugriffsrechte können selbstverständlich von Aufruf zu Aufruf wechseln (wechselnde Argumente).

1) Hierbei handelt es sich um die vierte Möglichkeit, einer Systemkomponente ein Zugriffsrecht einzuräumen.

2) Nun wird auch die Bedeutung der *f_potacc repr*-Kante (*representative of formal potential accesses*) in Abb. 2-20 klar: Sie repräsentiert ein *formales Zugriffsrecht* (*P* ist ein Dummy-ZR-Objekt).

3) *PICTSYN* kann lokal zu *SK* sein bzw. von *SK* importiert werden.

4) Auf die Systemgraph-Darstellung einer Benutzungsspezifikation wird später eingegangen.

5) Wird im Systemgraph durch eine *f_potacc*-Kante (*formal potential access*) beschrieben (siehe Abb. 2-21).

Es ist bereits zur Entwurfszeit entscheidbar, ob die Vergabe eines Zugriffsrechts hinsichtlich des Zugriffsmodells konsistent ist, da zu diesem Zeitpunkt bereits die Systemstruktur festgeschrieben wird. Ein *aktueller* Zugriff, also die eigentliche aktuelle Ausnutzung eines Zugriffsrechts (im Quelltext der Implementation einer Systemkomponente), ist hinsichtlich seiner Konsistenz ebenfalls statisch, nämlich zur Implementierungszeit überprüfbar.

Die bisher ausschließlich für abstr. Datenobjekte als ZR-Objekte betrachtete Zugriffsrecht-Beziehung ließe sich in vereinfachter Form prinzipiell auch auf einfache Datenobjekte ausdehnen. Dadurch würde die Datensicherheit und Zugriffstransparenz insgesamt erhöht werden. Diese mögliche Erweiterung der Zugriffsrecht-Beziehung bleibt jedoch im folgenden unberücksichtigt. Ein einfaches Datenobjekt ist somit nur für die vereinbarende Systemkomponente zugreifbar.

2.2.3.3 BENUTZUNGSRECHT-BEZIEHUNG

Es erwies sich als notwendig, Zugriffe auf abstrakte Datenobjekte als Informationsbehälter zu kontrollieren. Als wirksamer Schutzmechanismus bot sich die Vergabe von Zugriffsrechten an. Ist nun auch die Benutzung der (generischen) abstrakten Datentypen und der (parametrisierten) Funktionsklassen zu kontrollieren? Zunächst erscheint eine derartige Kontrolle für Muster von Datenobjekten bzw. algorithmische Elemente als überflüssig.

Daß trotzdem eine *Benutzungskontrolle* vorzusehen ist, illustriere folgendes Beispiel (vgl. Abb. 2-31):

Eine Systemkomponente *SK* benötigt zu ihrer Realisierung ein ADO *ADO1* von einem bestimmten Typ als lokale Ressource. Da dieser Typ bereits in einem anderen Teilsystem benötigt und dort als ADT *ADT* eingeführt wurde, möchte man gerne bei der Deklaration von *ADO1* auf diesen ADT zurückgreifen. Besitzt nun jedoch *ADT* selbst Zugriffsrechte, z.B. bzgl. eines ADO's *ADO2*, müssen auch *SK* dieselben Zugriffsrechte (mit den entsprechenden ZR-Attributen) eingeräumt sein. Ansonsten würde *ADO1* als *ADT*-Objekt beispielsweise auf *ADO2* zugreifen (*ADO1* besitzt virtuelles Zugriffsrecht bzgl. *ADT*, $\Rightarrow$ -Kante), ohne daß *SK* entsprechendes Zugriffsrecht besitzt. Dies widerspricht jedoch dem Prinzip der Zugriffstransparenz des Zugriffsmodells.

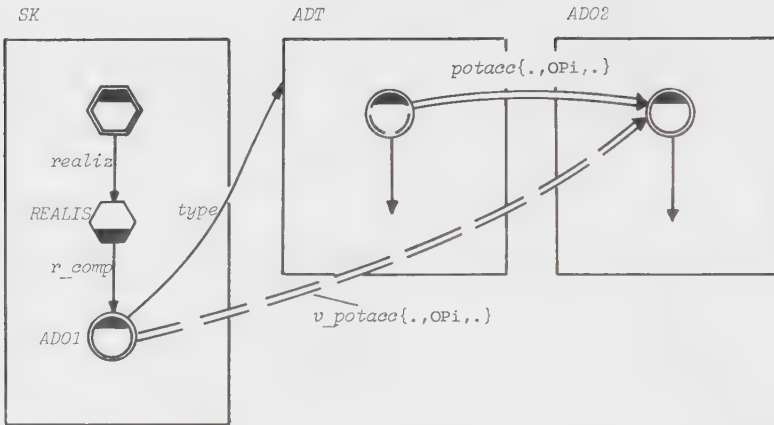


Abb. 2-31: Mangelnde Zugriffstransparenz

Um also nicht die Regeln der Zugriffsrecht-Vergabe im Zugriffsmodell zu unterlaufen, ist vor der aktuellen Benutzung eines (G)ADT's / einer (P)FK zu überprüfen, ob die Zugriffsrechte der benutzten Systemkomponente *SK'* durch die Zugriffsrechte der benutzenden Systemkomponente *SK* abgedeckt sind. Um nun nicht diese Konsistenzprüfung in der Realisierung von *SK* mehrmals (bei jeder aktuellen Benutzung) durchführen zu müssen, ist *SK* generell ein *Benutzungsrecht (BR)* bzgl. *SK'* einzuräumen, nachdem sich die Konsistenz mit den Regeln für die Zugriffsrecht-Vergabe als gegeben erwies.¹⁾ Zwischen *SK* (dem *BR-Subjekt*) und *SK'* (dem *BR-Objekt*) besteht dann eine *Benutzungsrecht-Beziehung (potentielle Benutzung)*. Erst dann kann *SK'* von *SK* *aktuell* benutzt werden.

Analog ZR-Attributen spezifizieren BR-Attribute die Operationen bzw. Funktionen, auf die sich die Benutzung des BR-Objekts beschränkt.²⁾

¹⁾ Dadurch wird auch die Benutzungsstruktur transparenter.

²⁾ BR-Attribute werden in erster Linie aus Orthogonalitätsgründen eingeführt, obwohl prinzipiell auch die Vergabe eines "generellen" Benutzungsrechts (ohne Attribute) genügen würde.

Beispielsweise könnte einem ADT *ADT* (u.a.) Benutzungsrecht bzgl. eines GADT's *GADT* und einer PFK *PFK* eingeräumt werden, wobei nur bestimmte Operationen/Funktionen bekanntgemacht werden:

```
adt ADT
  external interface
    type operations
      :
    access / use rights
      :
      gadt GADT by OP3, OP6;
      pfk PFK by FK2, FK3;
      :
  end external interface
  :
```

Abb. 2-32

Analog dem Zugriffsbereich ist der *Benutzungsbereich* einer Systemkomponente *SK* anzugeben. Wir gehen von der implizit gegebenen Benutzungsstruktur aus: Alle Systemkomponenten, die analog der Blockstruktur-Regelung in der Realisiert-Hierarchie für *SK* "sichtbar" sind, sind für *SK* auch benutzbar.¹⁾ Das Prinzip des information hidings ist somit auch für (generische) abstrakte Datentypen und (parametrisierte) Funktionsklassen gewahrt.

Im Zusammenhang mit der Definition des Zugriffsbereichs wurde nun die Menge der potentiellen ZR-Objekte aufgrund der Vererbungsregel weiter eingeschränkt. Der Grund für die Vererbungsregel lag im Erstreben von Zugriffstransparenz: Jede Systemkomponente sollte selbst alle Zugriffsrechte besitzen, die sämtlichen lokalen Systemkomponenten maximal eingeräumt werden. Da (generische) abstrakte Datentypen und (parametrisierte) Funktionsklassen keinen Zustandsänderungen unterliegen, also nur "lesend" verwendet werden, erscheint eine analoge Regel zur Vererbung von Benutzungsrechten nicht not-

¹⁾ Mit einer Ausnahme: (generische) abstrakte Datentypen sollten sich weder direkt noch indirekt rekursiv benutzen dürfen.

wendig: Eine Systemkomponente kann ein Benutzungsrecht besitzen, obwohl ihr Vater dieses Benutzungsrecht nicht besitzt. Natürlich darf dieses Benutzungsrecht nur eingeräumt werden, wenn die Zugriffsrechte (mit den ZR-Attributen) des BR-Objekts durch die des BR-Subjekts abgedeckt sind.

Der *Benutzungsbereich*¹⁾ einer Systemkomponente *SK* umfaßt somit

- a) die (G)ADT-/ (P)FK-Söhne von *SK*
- b) die (G)ADT-/ (P)FK-Brüder von *SK* (aber nicht deren Nachkommen)
- c) die (generischen) abstrakten Datentypen/ (parametrisierten) Funktionsklassen, die Brüder von direkten *SK*-Vorfahren sind.

Analog rekursiven Zugriffen erscheinen (direkt oder indirekt) rekursive Benutzungen zwischen (generischen) abstrakten Datentypen und (parametrisierten) Funktionsklassen als nicht sinnvoll, mit einer Ausnahme: Ist *SK* eine (P)FK, wird ihr Benutzungsbereich noch um *SK* selbst erweitert. Die Funktionen einer (P)FK sollten sich nämlich durchaus gegenseitig aufrufen dürfen. Ein typisches Beispiel dafür wäre ein Compiler für eine höhere Programmiersprache, der die Syntaxanalyse anhand des Verfahrens des rekursiven Abstiegs /SCH75a/ durchführt. Dabei bietet es sich an, die Analyseprozeduren, die zu den nichtterminalen Symbolen der zugrundegelegten Grammatik korrespondieren, als Funktionen einer FK *SYN_PROZ* zu spezifizieren. Charakteristisch für das verwendete Verfahren rufen sich *SYN_PROZ*-Funktionen gegenseitig auf: *SYN_PROZ* muß also sich selbst benutzen dürfen.

Die Benutzung eines direkten (P)FK-Vorfahren durch *SK* erscheint jedoch als nicht notwendig.

Einer beliebigen Systemkomponente *SK* kann also ein Benutzungsrecht bzgl. eines (G)ADT's/einer (P)FK eingeräumt werden, die

- a) im Benutzungsbereich von *SK* liegt. und
- b) bzgl. eines beliebigen ADO's *ADO* Zugriffsrecht (mit entsprechenden ZR-Attributen) nur dann besitzt, wenn auch *SK* dieses Zugriffsrecht (mit mindestens denselben ZR-Attributen) besitzt.

Jede Systemkomponente besitzt bzgl. ihrer (G)ADT-/ (P)FK-Söhne automatisch Benutzungsrecht (*implizites Benutzungsrecht*): Obige Bedin-

¹⁾ Invers dazu ist der *Benutzbarkeitsbereich*.

gung b) ist aufgrund der Vererbungsregel für Zugriffsrechte trivialerweise erfüllt.

Bzgl. allen anderen (generischen) abstrakten Datentypen/(parametrisierten) Funktionsklassen des Benutzungsbereichs kann ein Benutzungsrecht (mit BR-Attributen) nur explizit eingetragen werden (*explizites Benutzungsrecht*) (vgl. Abb. 2-32).

Implizite und explizite Benutzungsrechte treten nur im Zusammenhang mit (parametrisierten) Funktionsklassen und datenorientierten Systemkomponenten auf, deren Typ durch direkte Angabe der Typdefinition deklariert wird. Analog zu den Zugriffsrechten ließen sich für alle anderen BR-Subjekte *virtuelle Benutzungsrechte* betrachten. Diese sind jedoch im folgenden nicht von Bedeutung.

Der Eintrag eines Benutzungsrechts bewirkt kein Übertragen der Zugriffsrechte des BR-Objekts auf das BR-Subjekt.¹⁾

Benutzungsrechte treten auch auf Parameterposition auf: generische abstr. Datentypen können *formale BR-Parameter* als generische Parameter besitzen, die bei aktueller Benutzung durch entsprechende *aktuelle BR-Parameter* abzusättigen sind. Dies wird später noch detaillierter ausgeführt.

Explizite Benutzungsrechte werden im Systemgraph analog expliziten Zugriffsrechten beschrieben (vgl. Beispiel Abb. 2-32):²⁾

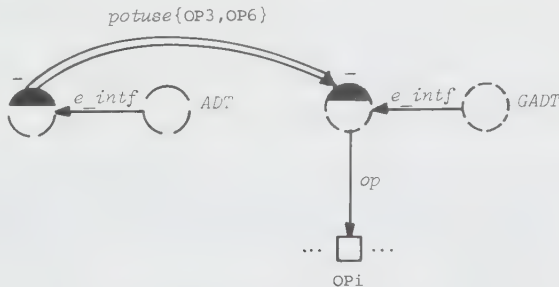


Abb. 2-33

Ein Benutzungsrecht wird durch eine *potuse*-Kante (potential use) ausgedrückt.

1) Das BR-Subjekt besitzt die Zugriffsrechte des BR-Objekts bereits, da sonst das Benutzungsrecht unzulässig wäre.

2) Implizite Benutzungsrechte werden nicht eigens dargestellt.

Die bisher ausschließlich für (generische) abstrakte Datentypen/ (parametrisierte) Funktionsklassen als BR-Objekte betrachtete Benutzungsrecht-Beziehung ließe sich in orthogonaler Weise auch auf einfache Datentypen ausdehnen.

Mit der vorgestellten Benutzungsrecht-Vergabe ist ein sinnvoller Kompromiß zwischen einer uneingeschränkten Benutzung (würde die Regeln der Zugriffsrecht-Vergabe unterlaufen) und einer Benutzungsrecht-Vergabe unter Berücksichtigung der Vererbung von Rechten (wäre zu restriktiv) gefunden.

Die Konsistenz der Benutzungsrecht-Vergabe ist (analog der Zugriffsrecht-Vergabe) zur Entwurfszeit entscheidbar. Jede *aktuelle* Benutzung, also das Ausnutzen eines Benutzungsrechts, ist hinsichtlich ihrer Konsistenz mit der Benutzungsrecht-Vergabe (spätestens) zur Implementierungszeit überprüfbar.

2.2.4 DETAILASPEKTE

In diesem Abschnitt werden einige Details des Strukturmodells präzisiert, die im letzten Abschnitt nur angedeutet wurden.

2.2.4.1 PARAMETERARTEN BEI GENERISCHEN ABSTRAKTEN DATENTYPEN

Als Arten generischer Parameter¹⁾ eines GADT's sind *Objekt-Parameter* und *Typ-Parameter* vorgesehen.

a) Objekt-Parameter

Zu dieser Parameterart zählen einerseits einfache Datenobjekte (*EDO-Parameter*).

Beispiele:

Die Parameter des GADT's *GT_ASSMEM* in Abb. 2-7 bzw. der Parameter zur Spezifikation der maximalen Kellertiefe eines generischen Kellertyps sind EDO-Parameter.

¹⁾ Dies sind Entwurfszeit-Parameter.

Zum anderen sind auch ZR-Parameter (*ADO-Parameter*) Objekt-Parameter.¹⁾ In einer Typkonkretisierung wird jeder formale ZR-Parameter durch einen Aktualparameter ersetzt, so daß in den GADT-Operationen (im Gegensatz zu den PFK-Funktionen) sämtliche aktuellen Zugriffe hinsichtlich des ZR-Objekts noch zur Entwurfszeit fixiert werden.

Beispiel:

Ein GADT *GADT* besitzt ein formales Zugriffsrecht bzgl. des (formalen) ADO's *F_ADO* (vom ADT *ADT1*):²⁾

```
gadt GADT (...; ado F_ADO : ADT1 by ..., OPi, ...; ...)
  external interface
  :
```

Abb. 2-34

Wird nun ein ADT *ADT2* mittels *GADT* deklariert, ist das formale Zugriffsrecht durch ein aktuelles Zugriffsrecht (hier ZR-Objekt *ADO* vom ADT *ADT1*) abzusättigen:

```
(g)adt ADT2 : GADT (...; ADO; ...) provides .. OPj, ...;
```

Abb. 2-35

Dieses Zugriffsrecht ist somit *ADT2* (über alle aktuellen Benutzungen von *ADT2* hinweg) unveränderbar eingeräumt. An der Stelle der Deklaration von *ADT2* müssen natürlich die ZR-Attribute bzgl. *ADO* die ZR-Attribute bzgl. *F_ADO* abdecken. *ADT2* erhält bzgl. *ADO* dieselben ZR-Attribute wie *GADT* bzgl. *F_ADO*.

-
- 1) Hinweis: Der Zugreifbarkeitsbereich eines formalen ZR-Parameters entspricht dem eines lokalen ADO's des GADT's. Somit ist ein formales Zugriffsrecht auch an die lokalen Systemkomponenten des GADT's vererbbar.
- 2) Die ZR-Attribute könnten auch fehlen: *F_ADO* dürfte dann nur in Wertzuweisungen bzw. (Un-)Gleichheitstests auftreten.

Darstellung im Systemgraph:

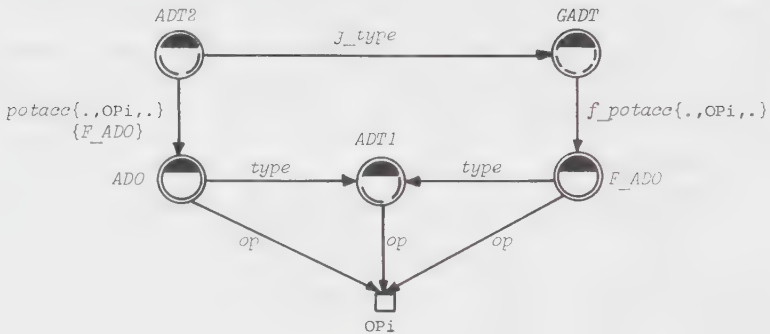


Abb. 2-36: formale/aktuelle ZR-Parameter

Die Darstellung eines formalen Zugriffsrechts durch eine f_potacc -Kante und die eines aktuellen Zugriffsrechts durch eine $potacc$ -Kante ist bereits bekannt. Der Attributwert $\{F_ADO\}$, mit dem die $potacc$ -Kante zusätzlich markiert ist, stellt die Zuordnung aktuelles Zugriffsrecht $\rightarrow$ formales Zugriffsrecht her: Das aktuelle Zugriffsrecht bzgl. ADO sätigt hier (nur) das formale Zugriffsrecht bzgl. F_ADO ab. Im allgemeinen ist der Attributwert eine Menge mit mehreren Elementen: Das aktuelle Zugriffsrecht kann nämlich gleichzeitig Aktualparameter zu mehreren Formalparametern sein.

Prinzipiell müssen (analog den Funktionen einer PFK) auch die Operationen datenorientierter Systemkomponenten mit formalen ZR-Parametern versehen werden können, für die zur Laufzeit aktuelle Zugriffsrechte übergeben werden.¹⁾ Ähnlich einer Benutzungsspezifikation für parametrisierte Funktionsklassen müßten sämtliche als Aktualparameter in Frage kommenden Zugriffsrechte zur Entwurfszeit bereits fixiert werden. Diese Maßnahme bleibt jedoch im folgenden aus Gründen der Übersichtlichkeit unberücksichtigt.

1) Ein Beispiel dafür sind die Operationen eines Kellerobjekts, die abstrakte Datenobjekte entgegennehmen bzw. zurückgeben.

b) Typ-Parameter

Unter diese Parameterart fallen zum einen einfache Datentypen (*EDT-Parameter*).

Ein Beispiel ist der GADT *GT_STACK*, der zur Spezifikation des Typs der Kellerelemente den formalen EDT-Parameter *T_ELEM* führt:¹⁾

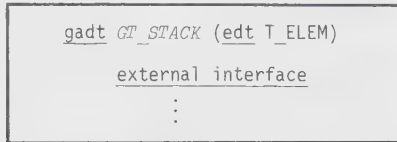


Abb. 2-37

Ein entsprechender Aktualparameter könnte z.B. *INTEGER* oder *POTATOE* sein:



Abb. 2-38

Darstellung im Systemgraph:

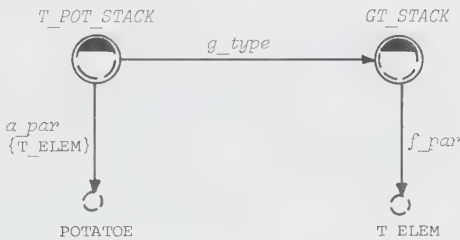


Abb. 2-39: formaler/aktueller EDT-Parameter

Nun könnte andererseits ein generischer Kellertyp benötigt werden, der zur Erzeugung von Kellern mit "komplexeren" Kellerelementen dient, z.B. Kellern von Teil-Symbollisten im Compiler,

¹⁾ *T_ELEM* müßte evtl. noch exakter spezifiziert werden, z.B. als skalarer Typ, Unterbereichstyp etc..

von Baumstrukturen etc.. Die Kellerelemente wären also nicht mehr von einem einfachen Datentyp, sondern von einem ADT, so daß also der formale generische Typ-Parameter T_ELEM aktuell mit einem Benutzungsrecht, z.B. bzgl. des ADT's ADT , abzusätigen ist (ADT -Parameter bzw. BR -Parameter):

```

gadt GT_STACK (adt T_ELEM)

    external interface
    :
    :

```

Abb. 2-40

In GT_STACK können nun abstrakte Datenobjekte vom ADT T_ELEM deklariert werden, ohne daß jedoch auf ihnen Operationen erklärt wären: Sie können, nur in Wertzuweisungen bzw. (Un-)Gleichheitstests auftreten.¹⁾ Dies genügt i.a. für die Realisierung eines Kellers.

Sollten jedoch gewisse GT_STACK -Operationen bestimmte Informationen in das oberste Kellerelement schreiben bzw. aus ihm lesen können, müssen die Zugriffsoperationen jedes aktuellen ADT-Parameters ADT bekannt sein. Die Kandidatenmenge für die Aktualparametrisierung ist deshalb einzuschränken: Es sind nur strukturähnliche abstrakte Datentypen als aktuelle ADT-Parameter zulässig. Der GADT $GADT$, relativ zu dem die Strukturähnlichkeit besteht, ist in der Spezifikation des entsprechenden generischen Formalparameters anzugeben:²⁾

```

gadt GT_STACK (adt T_ELEM : GADT by ..., OPi, ...)

    external interface
    :
    :

```

Abb. 2-41

- 1) Strenggenommen könnte deshalb in diesem Fall die Spezifikation des Formalparameters mit der eines EDT-Parameters zu "typ T_ELEM " zusammengefaßt werden.
- 2) GT_STACK muß ein explizites Benutzungsrecht bzgl. $GADT$ besitzen. Nach "by" sind die $GADT$ -Operationen aufgelistet, von denen in der Realisierung von GT_STACK Gebrauch gemacht werden darf.

Darstellung im Systemgraph:

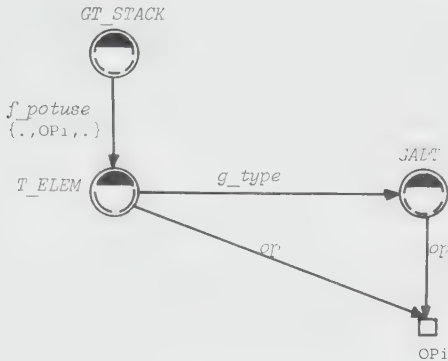


Abb. 2-42: formale BR-Parameter¹⁾

Bei einer aktuellen Benutzung von *GT_STACK* wird das formale Benutzungsrecht (mit dem BR-Objekt *T_ELEM*) durch ein aktuelles (mit dem BR-Objekt *ADT*) ersetzt:

adt *T_ADT_STACK* : *GT_STACK* (*ADT*) provides ...;

Abb. 2-43

ADT muß dabei eine Ausprägung von *GADT* sein, an der Stelle der Deklaration von *T_ADT_STACK* benutzt werden dürfen und mindestens die in der Spezifikation des formalen BR-Objekts *T_ELEM* aufgeführten Operationen besitzen.²⁾

Die Systemgraph-Darstellung der Aktualparametrisierung eines BR-Parameters ist analog zu der eines ZR-Parameters (vgl. Abb. 2-36).

¹⁾ Die Darstellung einer Deklaration eines formalen BR-Objekts (*T_ELEM*) lehnt sich also an die einer üblichen ADT-Deklaration mittels *GADT* an.

²⁾ Im Zuge einer Typ-Konkretisierung kann bekanntlich der Operationensatz des *GADT*'s abgemagert werden.

2.2.4.2 BENUTZUNGSSPEZIFIKATION

In einer Benutzungsspezifikation werden einer zu benutzenden PFK Zugriffsrechte in Form aktueller ZR-Parameter übertragen, um dadurch alle dynamisch möglichen Zugriffe einzugrenzen (vgl. Abschn. 2.2.3.2). Es ist noch die Systemgraph-Darstellung einer Benutzungsspezifikation anhand des Beispiels in Abb. 2-30 nachzutragen:

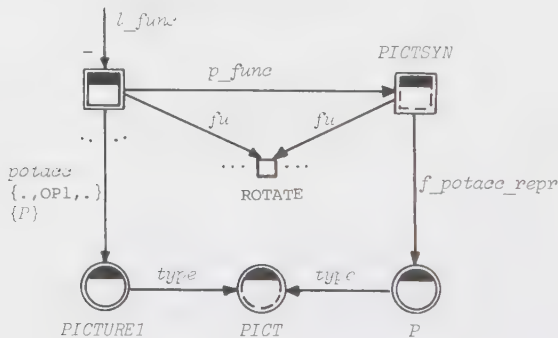


Abb. 2-44: Benutzungsspezifikation

Die Tatsache, daß das Ergebnis der Aktualparametrisierung von *PICTSYN* eine Funktionsklasse ist, wird durch eine *p_func*-Kante (parameterized function class) vom -Knoten¹⁾ zum Knoten mit Markierung *PICTSYN* ausgedrückt.

Die in der Benutzungsspezifikation aktuell eingeräumten Zugriffsrechte sind durch *potacc*-Kanten ausgedrückt, die vom Knoten für die virtuelle äußere Schnittstelle der FK zu den Knoten der ZR-Objekte führen. Der Markierungs-Attributwert $\{.,OPl,.\}$ beschreibt die aktuell eingeräumten ZR-Attribute, während $\{P\}$ die Zuordnung zum entsprechenden Formalparameter herstellt.

Die *P* aktuell zugeordneten ZR-Objekte müssen von demselben ADT wie *P*, also vom Typ *PICT*, sein.

Der derart spezifizierten FK werden sämtliche benutzbaren *PICTSYN*-Funktionen (in ihre virtuelle äußere Schnittstelle) übertragen. Dieser Sachverhalt wird durch die *fu*-Kanten festgehalten.

1) Es sei erinnert:  ist eine abkürzende Notation für 

2.2.4.3 DEKLARATIONEN DATENORIENTIERTER SYSTEMKOMPONENTEN

Es können mehrere datenorientierte Systemkomponenten, die von demselben expliziten (G)ADT sein sollen und dieselben Operationen erhalten sollen, gemeinsam in einer Liste deklariert werden, beispielsweise:

ado STACK1, STACK2, STACK3 : T_STACK provides .., OPi, ..;

Abb. 2-45

Zur Strukturierung abstr. Datenobjekte werden die im Bereich des Programmierens-im-Kleinen üblichen Datenstrukturierungs-Möglichkeiten vorgesehen. Abb. 2-46 zeigt die Deklaration eines ADO-Felds (vgl. Beispiel in Abschnitt 2.2.3.1):

ado NMPAGE : array [1:10] of T_NMPAGE provides .., OPi, ..;

Abb. 2-46

2.2.4.4 AKTUELLE BENUTZUNG

Besitzt eine Systemkomponente *SK* ein Benutzungsrecht bzgl. eines (G)ADT's bzw. einer PFK *SK'*, kann *SK'* in der äußeren und der inneren Schnittstelle von *SK* auch *aktuell benutzt* werden, also von *SK'* tatsächlich Gebrauch gemacht werden.¹⁾

Ein ADT kann von *SK* zur Spezifikation des Typs eines ADO's aktuell benutzt werden, und zwar

- a) in der äußeren Schnittstelle von *SK* als Typ
 - 1) eines formalen ZR-Objekts zu *SK*, falls *SK* ein GADT oder eine PFK ist,
 - 2) eines formalen ZR-Objekts einer *SK*-Operation/-Funktion.
- b) in der inneren Schnittstelle von *SK* als Typ eines mittels explizitem ADT deklarierten ADO's.

¹⁾ Das Benutzungsrecht kann auch formal sein, wenn nämlich *SK'* formales BR-Objekt eines GADT's ist.

Ein GADT kann aktuell benutzt werden

- a) in der äußeren Schnittstelle von *SK* als generischer Typ eines formalen BR-Objekts zu *SK*, falls *SK* ein GADT ist.
- b) in der inneren Schnittstelle von *SK* als (generischer) Typ eines ADO's bzw. ADT's.

Die aktuelle Benutzung einer PFK ist auf Benutzungsspezifikationen beschränkt.

Aktuelle Zugriffe und aktuelle Aufrufe von Funktionen einer (P)FK interessieren erst in einem späteren Zusammenhang.

Für einige Funktionen des vorgestellten Software-Arbeitsplatzes ist die Kenntnis darüber nützlich, ob in der äußeren bzw. der inneren Schnittstelle einer Systemkomponente *SK* von einem (formalen) Benutzungsrecht bzgl. eines (G)ADT's *SK'* überhaupt aktuell Gebrauch gemacht wird. Dazu wird die *actuses*-Kante (*actual uses*) eingeführt, die vom Knoten für die äußere Schnittstelle bzw. für die Realisierung von *SK* zur äußeren Schnittstelle von *SK'* gezogen wird, falls *SK'* mindestens einmal aktuell benutzt wird. Diese "technische" Kante führt als Werte von Markierungs-Attributen zusätzlich die Operationen des (G)ADT's auf, auf die sich die aktuellen Benutzungen insgesamt beschränken.

Beispiele:

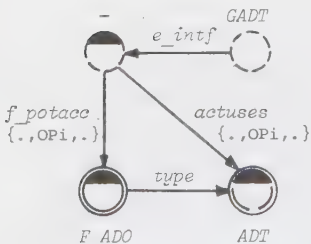


Abb. 2-47: aktuelle Benutzung eines ADT's (als Typ eines formalen ZR-Objekts)¹⁾

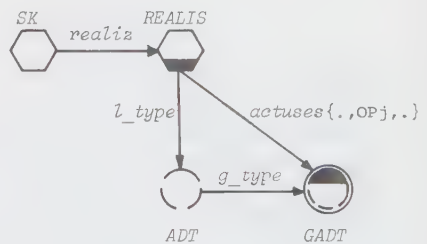


Abb. 2-48: aktuelle Benutzung eines GADT's (in einer ADT-Deklaration)²⁾

1) *GADT* muß bzgl. *ADT* explizites Benutzungsrecht besitzen.

2) *SK* muß bezgl. *GADT* (explizites oder implizites) Benutzungsrecht besitzen.

2.2.4.5 NAMEN VON SYSTEMKOMPONENTEN

Bisher wurde implizit unterstellt, daß alle Systemkomponenten im zu entwickelnden System durch einen einzigen Bezeichner eindeutig identifizierbar sind.

Diese Vorgehensweise würde es nötig machen, im Falle eines einzelnen Entwicklers sämtliche bereits vergebenen Bezeichner zu kennen bzw. im Falle eines Entwicklungsteams entsprechende Absprachen zu treffen. Zudem würde das Prinzip des information hidings mißachtet, da die Bezeichner lokaler Systemkomponenten, Realisierungs-Details also, für die übrigen Systemkomponenten bekannt sein müßten.

Zur eindeutigen Identifikation einer Systemkomponente ist deshalb bei ihrem angewandten Auftreten die Angabe des gesamten Selektorpfs nötig, der sich aus den Bezeichnern ihrer direkten Vorfahren in der Realisiert-Hierarchie (von der Systemwurzel bis zur Systemkomponente selbst) zusammensetzt.

Beispiel:

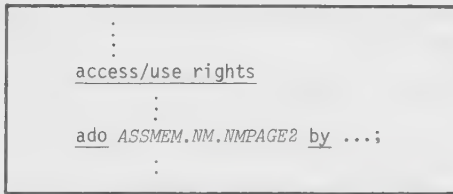


Abb. 2-49

Man könnte natürlich die Möglichkeit vorsehen, diese Form der Referenzierung innerhalb der Spezifikation der zugreifenden/benutzenden Systemkomponente durch einen einzigen, systemglobal eindeutigen Bezeichner abzukürzen.

Im folgenden werden jedoch Systemkomponenten aus Lesbarkeitsgründen wieder über einen einzigen Bezeichner identifiziert.

2.2.5 REALISIERUNGS-VARIANTEN

Es ist - wie die Praxis zeigt - oft sinnvoll, mehrere *Realisierungs-Varianten* zu einer Systemkomponente zuzulassen. Jede Realisierungs-Variante spannt dabei eine alternative *Teilsystem-Variante* auf.

Es gilt somit im folgenden auseinanderzuhalten: Unter einem *durch eine Systemkomponente SK aufgespannten Teilsystem* bzw. einem *Teilsystem zu einer Systemkomponente SK* werde die äußere Schnittstelle

von *SK* zusammen mit allen Teilhierarchien verstanden, die durch die Realisierungs-Varianten von *SK* aufgebaut werden. Dagegen werde unter einer *durch eine bestimmte Realisierungs-Variante von SK aufgespannten Teilsystem-Variante* die äußere Schnittstelle von *SK* zusammen mit der Teilhierarchie verstanden, die durch diese bestimmte Realisierungs-Variante aufgebaut wird.

Jede Realisierungs-Variante ist über einen eigenen Bezeichner referenzierbar. Realisierungs-Varianten zu einer Systemkomponente müssen verschieden bezeichnet sein.

Beispiel:

In Abschnitt 2.2.2.4 wurde der Namenspeicher *NM* des Assoziativspeichers *ASSMEM* als B-Baum realisiert, in Abschnitt 2.2.3.1 wurde eine alternative Realisierung als Tabelle mit Hashzugriff aufgeführt. *NM* wäre also folgendermaßen zu spezifizieren:

```
ado NM

  external interface
    :
  end external interface

  ado-realization B_TREE_VARIANT
    :
  end ado-realization B_TREE_VARIANT

  ado-realization TABLE_VARIANT
    :
  end ado-realization TABLE_VARIANT

end ado NM
```

Abb. 2-50

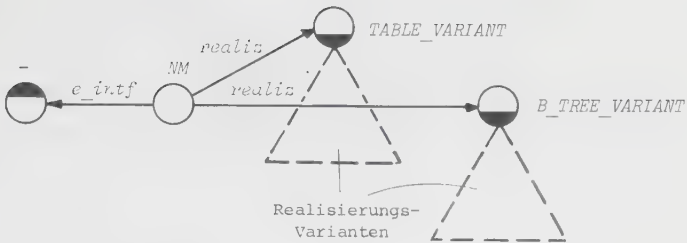


Abb. 2-51: Systemgraph-Darstellung von Realisierungs-Varianten

Im Zuge der Systemgenerierung wird aus der Sammlung der Realisierungs-Varianten zu einer Systemkomponente jeweils eine Variante aufgrund bestimmter Eigenschaften (z.B. laufzeitoptimale Variante, speicherplatzoptimale Variante) ausgewählt. Die Charakteristika einer Realisierungs-Variante werden dazu in *Realisierungs-Attributen* beschrieben, z.B.

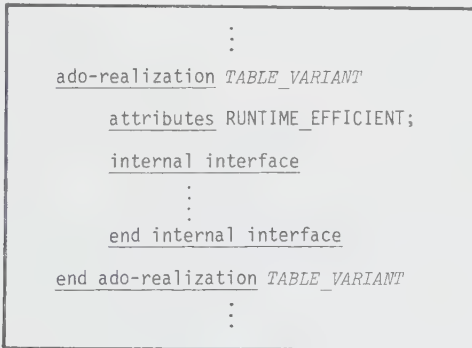


Abb. 2-52

Die möglichen Realisierungs-Attribute sind vom Benutzer frei definierbar.

Im Systemgraph werden die Attribute zu einer Realisierung als Attribute des entsprechenden Realisierungs-Knoten eingebracht:



Abb. 2-53

KAPITEL 3: FORMALE BESCHREIBUNG DES SYSTEMGRAPHEN

In diesem Kapitel wird der Systemgraph mit seinen wichtigsten Eigenschaften formal eingeführt. Der flüchtige Leser kann dieses Kapitel überblättern, ohne in den folgenden Kapiteln in Verständnis-Schwierigkeiten zu geraten.

3.1 GRAPHENTHEORETISCHE GRUNDLAGEN

Folgende Begriffe und Notationen werden im weiteren benötigt:

a) Seien ρ, ρ' binäre Relationen auf einer Basis(knoten)menge KNO, d.h. $\rho \subseteq \text{KNO} \times \text{KNO}$ und $\rho' \subseteq \text{KNO} \times \text{KNO}$.

$$1) \rho^+ := \{(kn, kn') \mid \exists n \in \mathbb{N} \ n > 0 \ \exists kn_1 \in \text{KNO} \ (1 \leq i \leq n) \\ (\forall 1 \leq j \leq n-1 \ ((kn_j, kn_{j+1}) \in \rho) \wedge kn = kn_1 \wedge \\ \wedge kn' = kn_n)\} \quad (\text{transitive Hülle bzgl. } \rho)$$

ρ^* analog.

$$\rho \cup \rho' := \{(kn, kn') \mid (kn, kn') \in \rho \vee (kn, kn') \in \rho'\} \\ (\text{Vereinigung})$$

2) Es gibt einen ρ -Pfad zwischen zwei Knoten $kn, kn' \in \text{KNO}$

$$: \iff (kn, kn') \in \rho^+$$

ρ heißt *zyklenfrei*

$$: \iff \forall kn \in \text{KNO} \ ((kn, kn) \notin \rho^+)$$

ρ besitzt *Waldstruktur*

$$: \iff \rho \text{ ist zyklenfrei} \wedge \\ \forall kn_1, kn_2, kn \in \text{KNO} \ ((kn_1, kn) \in \rho \wedge (kn_2, kn) \in \rho) \\ \implies kn_1 = kn_2$$

b) Seien $M_i \ (1 \leq i \leq n)$ Mengen, $M = \bigcup_{i=1}^n M_i$ und $m = (m_1, \dots, m_n) \in M$.

Für $1 \leq i \leq n$ sei $p_i(m)$ die *i. Projektion* von m , d.h. $p_i(m) = m_i$.

3.2 KONTEXTFREIE SYNTAX EINES SYSTEMGRAPHEN

Wir wollen nun die kontextfreie Syntax eines Systemgraphen formal definieren.

Ein *Systemgraph* $SG = (KNO, KNOTYPEN, knotyp, KNOMARKEN, knomark, KANMARKEN, (\rho_{kam})_{kam \in KANMARKEN})$ ist ein gerichteter, markierter Graph mit einer endlichen Menge von

- Knoten KNO , wobei die Knoten (auf irgendeine Weise) eindeutig bezeichnet sind.
- Knotentypen $KNOTYPEN = \{ado, adt, gadt, fk, pfk, ado_e_intf, adt_e_intf, \dots\}$. (Darstellung der Arten an Systemkomponenten und deren Bestandteile) und einer totalen Funktion $knotyp : KNO \rightarrow KNOTYPEN$.
Zu jedem Knotentyp $knt \in KNOTYPEN$ ist eine endliche, evtl. leere Menge von *Knotentyp-Attributen* definiert.
- Knotenmarkierungen $KNOMARKEN$ (benutzerdefinierte Bezeichner für Systemkomponenten und deren Bestandteile) und einer totalen Funktion $knomark : KNO \rightarrow KNOMARKEN \cup \{-\}$.¹⁾
- (evtl. leeren) binären Relationen ρ_{kam} auf der Knotenmenge KNO , also $\rho_{kam} \subseteq KNO \times KNO$. Jedes Knotenpaar $(kn1, kn2) \in \rho_{kam}$ ($kn1, kn2 \in KNO$) wird interpretiert als eine gerichtete, mit kam markierte Kante von $kn1$ nach $kn2$. kam ist ein Element der Kantenmarkierungen $KANMARKEN = \{e_intf, v_e_intf, realiz, r_comp, potacc, v_potacc, potuse, type, g_type, p_func, f_potacc, f_potuse, op, fu\}$ ²⁾ (syntaktische Kategorien von Beziehungen zwischen Systemkomponenten).
Zu jeder Kantenmarkierung $kam \in KANMARKEN$ ist eine endliche, evtl. leere Menge von *Kantenmarkierungs-Attributen* definiert.³⁾

Definition 3.1

Sei $knt \in KNOTYPEN$. kn ist ein knt -Knoten (kurz knt_kn) : $\longleftrightarrow$
 $knotyp(kn) = knt$

¹⁾ $KNOTYPEN$ kann auch als weitere Kategorie von Knotenmarkierungen gesehen werden.

²⁾ $KANMARKEN$ wird später noch um weitere Elemente ergänzt werden.

³⁾ Beispielsweise werden ZR-Attribute einer Zugriffsrecht-Beziehung als ein Kantenmarkierungs-Attribut notiert.

Beispiele:

ado_kn, ado_e_intf_kn.

Definition 3.2

Sei $knt \in KNOTYPEN$ ein Knotentyp bzw. $kam \in KANMARKEN$ eine Kantenmarkierung mit Knotentyp- (bzw. Kantenmarkierungs-)Attributen $attr_i (1 \leq i \leq n)$, wobei $attr_i$ Werte aus einer endlichen Wertemenge $attr_i\text{-WERTE} := \{attr_i\text{-wert}_1, \dots, attr_i\text{-wert}_{n_i}\}$ annehmen kann.¹⁾

Eine *aktuelle Attributierung* eines Knotens kn vom Typ knt bzw. einer Kante $ka = (kn_1, kn_2) \in \rho_{kam} (kam \in KANMARKEN)$ ist ein n -Tupel $(attr_1\text{-wert}_{j_1}, \dots, attr_n\text{-wert}_{j_n}) \in attr_1\text{-WERTE} \times \dots \times attr_n\text{-WERTE}$. *Aktueller $attr_i$ -Attributwert* von kn (bzw. ka):

$$\begin{aligned} attr_i(kn) &:= p_i((attr_1\text{-wert}_{j_1}, \dots, attr_n\text{-wert}_{j_n})) = \\ &= attr_i\text{-wert}_{j_i} \end{aligned}$$

Beispiel:

Sei $(kn_1, kn_2) \in \rho_{a_par}$ eine Kante mit Markierung a_par .

Das Attribut $fparam$ (aktuell ersetzte Formalparameter) besitzt als aktuellen Wert die Menge $\{F_PAR2, F_PAR3\}$.

Dann gilt:

$$fparam((kn_1, kn_2)) = \{F_PAR2, F_PAR3\}$$

3.3 KONSISTENZBEDINGUNGEN

Nicht jeder Systemgraph repräsentiert automatisch ein "wohlstrukturiertes" Softwaresystem. Es sind noch *Konsistenzbedingungen* *KONSBED* anzugeben, die nur solche Beziehungen zwischen Systemkomponenten garantieren, die den Strukturierungsprinzipien (hierarchische Systemstruktur, konsistente Zugriffsrecht- und Benutzungsrecht-Beziehung) genügen.²⁾

¹⁾ Ein Wert $attr_i\text{-wert}_{j_i}$ kann ein einzelner direkter Wert, aber auch eine Menge diskreter Werte sein. Beispiele: Der Attributwert $\{OP2, OP4\}$ für eine *potacc*-Kante oder $\{F_PAR\}$ für eine *a_par*-Kante.

²⁾ Notwendige Korrelationen zwischen der Markierung einer Kante und den Typen ihrer Quell- und Zielknoten (Zielknoten einer *potacc*-Kante muß beispielsweise stets ein Knoten vom Typ *ado* sein) seien als gegeben vorausgesetzt.

Bemerkung zu den im folgenden dargestellten Systemgraph-Ausschnitten:

-Kanten sollen den jeweils interessierenden Systemgraph-Kontext hervorheben;

-Kanten müssen existieren, -Kanten dürfen nicht existieren.

Ist eine Realisiert-Beziehung darzustellen, wobei die Art der Sohn-Systemkomponente irrelevant ist, wird anstelle der Kantenmarkierung "*r_comp / l_type / l_func*" (entsprechend den alternativen Arten der Sohn-Systemkomponente) abkürzend die Kantenmarkierung *l_sc* (local system component) verwendet.

In Kapitel 2 wurden öfters die Knoten für die (virtuelle) äußere Schnittstelle einer Systemkomponente und für die Systemkomponente selbst zur komprimierteren Darstellung bestimmter Sachverhalte zu  zusammengezogen. Im folgenden können analog der Knoten für eine Systemkomponente und der Knoten für eine ihrer Realisierungs-Varianten zu  zusammengefaßt sein. Soll zusätzlich der Knoten für die (virtuelle) äußere Schnittstelle in die abkürzende Darstellung eingehen, ist  der resultierende Knoten. In den Konsistenzbedingungen werden die *(v_)e_intf*- und *realiz*-Kanten, die aufgrund der Komprimierung entfallen, aus Übersichtlichkeitsgründen nicht aufgeführt.

3.3.1 KORREKTHEIT DER REALISIERT-HIERARCHIE

Eine erste Konsistenzbedingung (KB_HIER) dient zur Vermeidung einer rekursiven Definition von Systemkomponenten bzw. verhindert Doppeldeklarationen (jede Systemkomponente *SK* kann zu höchstens einer Vater-Systemkomponente lokal sein). Die in Abschnitt 2.2.3.1 beschriebene Realisiert-Hierarchie schließt im Systemgraph Konstellationen wie in Abb. 3-1 bzw. in Abb. 3-2 aus.

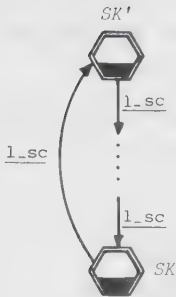


Abb. 3-1: Rekursive Definition

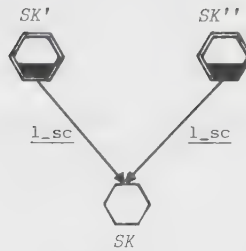


Abb. 3-2: Doppeldeklaration

Definition 3.3 (Konsistenzbedingung für korrekte Hierarchie)

$KB_HIER \equiv$

Die Knotenrelation ρ_{l_sc} ($= \rho_{r_comp} \cup \rho_{l_type} \cup \rho_{l_func}$)
besitzt Waldstruktur.¹⁾

Definition 3.4

Ein Systemgraph, in dem KB_HIER erfüllt ist, heißt *hierarchisch*.

3.3.2 KORREKTHEIT DER EXPLIZITEN ZUGRIFFSRECHT-BEZIEHUNG

Im folgenden werden Konsistenzbedingungen angegeben, die die Korrektheit der Vergabe expliziter Zugriffsrechte (*explizite Zugriffsrecht-Beziehung*) sichern.

Zunächst werden Bedingungen angeführt, die die einzigen zulässigen Situationen bzgl. der expliziten Zugriffsrecht-Beziehung beschreiben, anschließend (in ihrer Gesamtheit äquivalente) Bedingungen aufgestellt, die nicht zulässige Situationen ausschließen.

Nach Abschnitt 2.2.3.2 darf einer Systemkomponente SK ein explizites Zugriffsrecht bzgl. eines ADO's ADO nur dann eingeräumt wer-

¹⁾ ρ_{l_sc} besitzt deshalb nicht nur Baumstruktur, weil gleichzeitig mehrere noch nicht integrierte Teilsysteme existieren können.

den, wenn die Vater-Systemkomponente SK' bereits ein entsprechendes Zugriffsrecht (mit mindestens denselben ZR-Attributen) besitzt (Zugriffsrecht-Vererbung) bzw. ADO Bruder-Systemkomponente von SK ist.¹⁾

Nach der Zuteilung des Zugriffsrechts muß also genau eine der beiden Situationen aus Abb. 3-3 und Abb. 3-4 vorliegen.²⁾

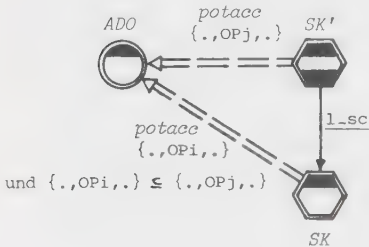


Abb. 3-3: Zugriffsrecht-Vererbung

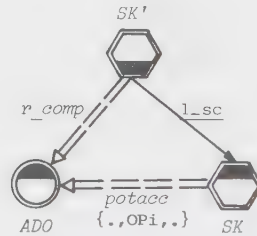


Abb. 3-4: Zugriff auf Bruder-Systemkomponente

Dieser Sachverhalt ist in der Konsistenzbedingung KB_KORR_POTZUGR der folgenden Definition formalisiert. Darin wird auch berücksichtigt, daß in der Situation von Abb. 3-4 das Zugriffsrecht nur eingetragen werden darf, falls dadurch kein Zugriffs-Zyklus von SK über ADO nach SK zurück entsteht.

- 1) Bei Zugriffsrechten als Aktualparameter ist für eine entsprechende Überprüfung die Systemkomponente mit den Formalparametern als ZR-Subjekt relevant.
- 2) Ein GADT kann ein formales Zugriffsrecht besitzen, wobei einer Sohn-Systemkomponente explizites Zugriffsrecht bzgl. des formalen ZR-Objekts eingeräumt wird. Solche Situationen wollen wir aus Übersichtlichkeitsgründen im folgenden nicht berücksichtigen.

Definition 3.5 (Konsistenzbedingung für korrekten potentiellen Zugriff)

$$\begin{aligned}
 \text{KB_KORR_POTZUGR} &\equiv \\
 &\vee \text{SKART_kn, SKART'_{kn}, ado_kn} \\
 &(((\text{SKART'_{kn}, SKART_kn}) \in \rho_{l_sc} \wedge \\
 &\quad \text{ka}^1) = (\text{SKART_kn, ado_kn}) \in \rho_{potacc}) \implies \\
 &((\text{ka}' = (\text{SKART'_{kn}, ado_kn}) \in \rho_{potacc} \\
 &\quad \wedge \text{paattr}(\text{ka})^2) \leq \text{paattr}(\text{ka}')) \\
 &\vee \\
 &((\text{SKART'_{kn}, ado_kn}) \in \rho_{r_comp} \\
 &\quad \wedge \text{paattr}(\text{ka}) \leq \text{op}(\text{ado_kn}) \\
 &\quad \wedge (\text{ado_kn, SKART_kn}) \notin \rho_{potacc}^+)) \\
 &\text{wobei: } \text{SKART, SKART}' \in \{\text{ado, (g)adt, (p)fk}^3\} \\
 &\quad \text{op}(\text{kn}) := \{\text{knomark}(\text{kn}') \mid (\text{kn, kn}') \in \rho_{op}\}
 \end{aligned}$$

□

Zugriffsrechte, die formale Zugriffsrechte in ADO-/(G)ADT-Deklarationen (jeweils mittels GADT) absättigen, müssen neben KB_KORR_POTZUGR auch der Forderung genügen, daß die ZR-Objekte der formalen und aktuellen Zugriffsrechte typäquivalent sind und die ZR-Attribute bzgl. des formalen ZR-Objekts durch die bzgl. des aktuellen ZR-Objekts überdeckt werden (vgl. Abb. 2-36).

Definition 3.6 (Konsistenzbedingung für korrekte potentielle Zugriffe auf Parameterposition)

$$\begin{aligned}
 \text{KB_PARAM_POTZUGR} &\equiv \text{a) } \wedge \text{b) mit} \\
 \text{a) } &\vee \text{gadt_kn, ado_kn, adt_kn, SKART_kn, ado_kn}' \\
 &((\text{ka} = (\text{gadt_kn, ado_kn}) \in \rho_{f_potacc} \\
 &\quad \wedge \text{knomark}(\text{ado_kn}) = F_{\text{ADO}} \\
 &\quad \wedge (\text{ado_kn, adt_kn}) \in \rho_{type}
 \end{aligned}$$

- 1) Die durch die rechts stehende Knotenrelation fixierte Kante wird mit ka bezeichnet.
- 2) Das Kantenmarkierungs-Attribut paattr (potential access attributes) besitzt als Wert die ZR-Attribute der durch die Kante ausgedrückten Zugriffsrecht-Beziehung.
- 3) Hierbei muß $\text{SKART}' = (\text{p})\text{fk} \implies \text{SKART} \neq \text{ado}$ gelten.

$$\begin{aligned}
 & \wedge (SKART_kn, gadt_kn) \in \rho_{g_type} \\
 & \wedge ka' = (SKART_kn, ado_kn') \in \rho_{potacc} \\
 & \wedge F_ADO \in f_param(ka') \\
 \Rightarrow & ((ado_kn', adt_kn) \in \rho_{type} \\
 & \wedge op(ado_kn) \leq op(ado_kn') \\
 & \wedge paattr(ka) \leq paattr(ka'))
 \end{aligned}$$

wobei: $SKART \in \{adt, gadt\}$
 $op(kn)$ analog Def. 2.8

b) ähnlich für Benutzungsspezifikation

□

Definition 3.7

Ein Systemgraph, in dem $KB_KORR_POTZUGR$ und $KB_PARAM_POTZUGR$ erfüllt sind, heißt bzgl. der expliziten Zugriffsrecht-Beziehung konsistent.

Im folgenden werden Konsistenzbedingungen eingeführt, die hinsichtlich der expliziten Zugriffsrecht-Beziehung unzulässige Vernetzungen (vor allem rekursive Definitionen und die Mißachtung des Prinzips des information hidings) ausschließen. Dieses Bündel von Bedingungen soll noch einmal die Charakteristiken des vorgeschlagenen Zugriffsmodells verdeutlichen.

Gemäß Abschnitt 2.2.3.2 darf eine Systemkomponente SK zur Vermeidung einer rekursiven Definition

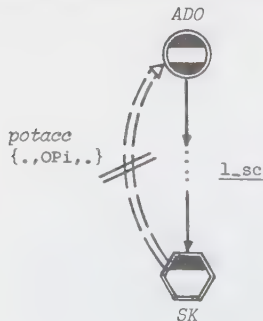


Abb. 3-5

nicht explizit auf ein ADO ADO zugreifen, falls ADO bzgl. der Realisiert-Beziehung Vorfahre von SK ist (Abb. 3-5).

Ebenso ist zu sichern, daß bei einem Zugriff von SK auf einen ADO-Bruder ADO dieser nicht wiederum (direkt oder indirekt) auf SK zugreift.

Die folgende Definition führt

die Bedingung KB_NICHTREK_POTZUGR ein, die die Rekursionsfreiheit garantiert.¹⁾

Definition 3.8 (Konsistenzbedingung zur Vermeidung nichtrekursiver potentieller Zugriffe)

$$\begin{aligned}
 \text{KB_NICHTREK_POTZUGR} &\equiv \\
 &\forall \text{ SKART_kn, ado_kn} \\
 &((\text{SKART_kn, ado_kn}) \in \rho_{\text{potace}} \Rightarrow \\
 &((\text{ado_kn, SKART_kn}) \in \rho_{\text{L_sc}}^* \wedge \\
 &(\text{ado_kn, SKART_kn}) \in \rho_{\text{potace}}^+))
 \end{aligned}$$

wobei: $\text{SKART} \in \{\text{ado}, (g)\text{adt}, (p)\text{fk}\}$ □

Die folgende Konsistenzbedingung (KB_INFID_POTZUGR) sichert die Wahrung des Prinzips des information hidings, daß nämlich lokale abstrakte Datenobjekte einer anderen datenorientierten Systemkomponente von außen her nicht zugreifbar sind.

Im Systemgraph darf es also keinen Teilgraph folgender Gestalt geben (ADO liegt außerhalb des Zugriffsbereichs von SK):

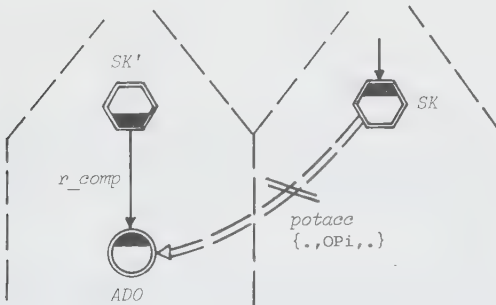


Abb. 3-6

¹⁾ Diese Bedingung schließt auch Zugriffsrechte aus, bei denen ZR-Subjekt und ZR-Objekt identisch sind.

Definition 3.9 (Konsistenzbedingung zur Wahrung von information hiding bei potentiellen Zugriffen)

KB_INFHD_POTZUGR $\equiv$

$$\begin{aligned} & \forall SKART_kn, ado_kn, SKART_kn \\ & ((SKART_kn, ado_kn) \in \rho_{l_sc} \wedge \\ & (SKART_kn, ado_kn) \in \rho_{potacc}) \Rightarrow \\ & (SKART_kn, SKART_kn) \in \rho_{l_sc}^+ \end{aligned}$$

wobei: $SKART \in \{ado, (g)adt, (p)fk\}$

$SKART' \in \{ado, (g)adt\}$

□

Bemerkung: Durch die Bedingung KB_INFHD_POTZUGR sind bereits explizite Zugriffsrechte einer Systemkomponente SK bzgl. eines lokalen ADO's ausgeschlossen:

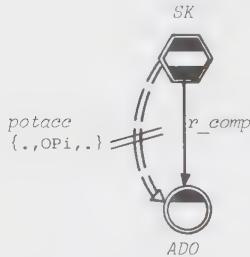


Abb. 3-7

In einem Systemgraph, in dem KB_NICHTREK_POTZUGR und KB_INFHD_POTZUGR gelten, können somit einer Systemkomponente SK nur explizite Zugriffsrechte eingeräumt werden, bei denen das ZR-Objekt

- ADO-Bruder von SK ist,
- ADO-Bruder eines Vorfahren von SK ist.

Im letzten Fall muß jedoch auch die Vater-Systemkomponente SK' von SK ein entsprechendes explizites Zugriffsrecht besitzen, was noch in einer Bedingung KB_VERERB_POTZUGR zu sichern ist. Die Formulierung dieser Bedingung ist aus der von KB_KORR_POTZUGR (Def. 3.5) extrahierbar.

Satz 3.10 hält fest, daß die Bedingung, die die einzigen zulässigen Situationen bzgl. der expliziten Zugriffsrecht-Beziehung be-

schreibt, äquivalent ist mit dem Satz an Bedingungen, die unzulässige Situationen ausschließen.

Satz 3.10 (ohne Beweis)

KB_KORR_POTZUGR erfüllt $\longleftrightarrow$

KB_NICHTREK_POTZUGR, KB_INFHD_POTZUGR und KB_VERERB_POTZUGR erfüllt.

Korollar 3.11

Ein Systemgraph ist bzgl. der expliziten Zugriffsrecht-Beziehung konsistent gdw. KB_NICHTREK_POTZUGR, KB_INFHD_POTZUGR, KB_VERERB_POTZUGR und KB_PARAM_POTZUGR erfüllt sind.

3.3.3 KORREKTHEIT DER EXPLIZITEN BENUTZUNGSRECHT-BEZIEHUNG

Die Korrektheit der expliziten Benutzungsrecht-Beziehung wird auf dem Systemgraph ebenfalls durch eine Reihe von Konsistenzbedingungen kontrolliert.

Analog der Bedingung KB_NICHTREK_POTZUGR aus Def. 3.8 kann eine Bedingung KB_NICHTREK_POTBEN zur Vermeidung rekursiver Definitionen von (generischen) abstrakten Datentypen und (parametrisierten) Funktionsklassen formuliert werden.^{1), 2)} Durch diese Bedingung

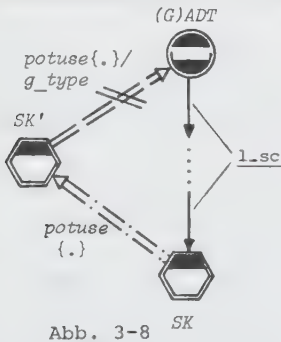


Abb. 3-8

sind auch Zyklen der Form von Abb. 3-8 auszuschließen (anstelle von (G)ADT kann auch eine PFK stehen). Zu verbieten sind auch rekursive Benutzungen zwischen Bruder-Systemkomponenten.

Ebenso lassen sich analog der Bedingung KB_INFHD_POTZUGR aus Def. 3.9 in einer Bedingung KB_INFHD_POTBEN (explizite) Be-

- 1) Bei Benutzungsrechten, die eine Aktualparametrisierung formaler Benutzungsrechte darstellen, ist für eine entsprechende Überprüfung die Systemkomponente als BR-Subjekt relevant, zu deren Deklaration die aktuellen Benutzungsrechte eingeräumt werden.
- 2) Eine Definition einer (P)FK durch direkte Rekursion ist jedoch zuzulassen.

nutzungsrechte ausschließen, die gegen das Prinzip des information hidings (hier bzgl. (generischer) abstrakter Datentypen und (parametrisierten) Funktionsklassen) verstoßen.

Da keine Vererbung von Benutzungsrechten vorgesehen ist, ist keine zu KB_VERERB_POTZUGR analoge Konsistenzbedingung nötig.

Die Vermeidung nicht-transparenter virtueller Zugriffe, die dem Zugriffsmodell widersprechen (siehe Abb. 2-31), wird durch die Konsistenzbedingung KB_VERERBVERTR_POTBEN (Verträglichkeit potentieller Benutzungen mit der Vererbungsregel) garantiert. Diese Bedingung verbietet einer Systemkomponente SK einen $(G)ADT$ $(G)ADT$ zu benutzen, falls $(G)ADT$ mehr (explizite/virtuelle) Zugriffsrechte besitzt als SK selbst, bzw. (positiv ausgedrückt), es sind nur folgende Situationen zulässig:

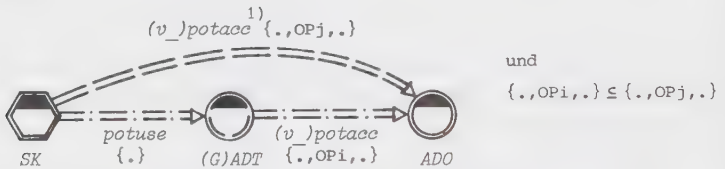


Abb. 3-9

Analoges gilt für eine (parametrisierte) Funktionsklasse $(P)FK$ anstelle von $(G)ADT$.²⁾

Definition 3.12 (Konsistenzbedingung für die Verträglichkeit potentieller Benutzungen mit der Vererbungsregel)

$$\begin{aligned} \text{KB_VERERBVERTR_POTBEN} &\equiv \\ &\forall \text{SKART_kn}, \text{SKART'_kn}, \text{ado_kn} \\ &(((\text{SKART_kn}, \text{SKART'_kn}) \in \rho_{\text{potuse}} \wedge \\ &\quad \text{ka} = (\text{SKART'_kn}, \text{ado_kn}) \in \rho_{(v)\text{potacc}}) \Rightarrow \end{aligned}$$

¹⁾ Die v_{potacc} -Kante ist in folgenden Situationen relevant: SK ist ein $(G)ADT$, der mittels eines $GADT$'s deklariert wird, wobei $(G)ADT$ ein aktuelles BR-Objekt ist.

²⁾ Zwischen $(P)FK$ und ADO kann dann nur eine potacc -Kante auftreten.

$$(ka' = (SKART_kn, ado_kn) \in \rho_{(v_)}potacc \\ \wedge paattr(ka) \subseteq paattr(ka'))$$

wobei: $SKART \in \{ado, (g)adt, (p)fk\}$
 $SKART' \in \{(g)adt, (p)fk\}$ □

Die Bedingung KB_NICHTREK_POTZUGR aus Def. 3.8 garantiert, daß eine Systemkomponente SK zur Vermeidung einer rekursiven Definition nicht auf ein ADO ADO explizit zugreift, falls ADO bzgl. der Realisiert-Hierarchie Vorfahre von SK ist.

KB_NICHTREK_POTZUGR deckt jedoch beispielsweise noch nicht die verbotenen Konstellationen einer rekursiven Definition aus Abb. 3-10

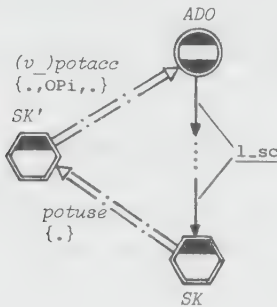


Abb. 3-10

ab: Einer Systemkomponente SK wird zur Realisierung ihrer Ressourcen ein Benutzungsrecht bzgl. SK' eingeräumt. Da jedoch SK' Zugriffsrecht bzgl. ADO , einem Vorfahren von SK , besitzt, würden bei entsprechendem aktuellen Zugriff/aktueller Benutzung die SK -Operationen/-Funktionen auf ADO operieren. Dies entspräche einer unerlaubten rekursiven Definition von ADO .

KB_NICHTREK_POTZUGR und KB_VERERBVERTR_POTBEN zusammen jedoch schließen solche Konstellationen aus: Nach KB_VERERBVERTR_POTBEN müßte SK in Abb. 3-10 (explizites) Zugriffsrecht bzgl. ADO besitzen, was jedoch durch KB_NICHTREK_POTZUGR ausgeschlossen wird.

Benutzungsrechte, die aktuell für formale BR-Parameter in ADO-/ (G)ADT-Deklarationen eingeräumt werden, müssen neben den bisher eingeführten Konsistenzbedingungen für Benutzungsrechte (evtl.) auch der Forderung genügen, daß die BR-Objekte der formalen und aktuellen Benutzungsrechte Ausprägungen desselben GADT's sind und das aktuelle BR-Objekt mindestens die gleichen Zugriffsoperationen bereitstellt wie das formale BR-Objekt. Dieser Sachverhalt läßt sich in einer zu KB_PARAM_POTZUGR a) analogen Bedingung KB_PARAM_POTBEN formalisieren.

Definition 3.13

Ein Systemgraph, in dem $KB_NICHTREK_POTBEN$, KB_INFHID_POTBEN , $KB_VERERBVERTR_POTBEN$ und KB_PARAM_POTBEN erfüllt sind, heißt bzgl. der expliziten Benutzungsrecht-Beziehung konsistent.

3.3.4 ZUGRIFFSRECHT-KONSISTENZ

In den Abschnitten 3.3.2 und 3.3.3 wurden Bedingungen für eine korrekte explizite Zugriffs- und Benutzungsrecht-Beziehung angegeben. Es gilt nun zu klären, inwieweit dadurch auch die Konsistenz hinsichtlich des Zugriffsmodells schlechthin (Zugriffsrecht-Konsistenz) garantiert ist.

Die Zugriffsrecht-Konsistenz ist dann gewahrt, wenn $KB_KORR_POTZUGR$ auch bei Ausdehnung auf virtuelle Zugriffsrechte gilt:

Definition 3.14 (Konsistenzbedingung für die Konsistenz potentieller

$KB_KONS_POTZUGR \equiv$ ler Zugriffe)
 $KB_KORR_POTZUGR$, wobei zwischen SK und ADO auch eine virtuelle Zugriffsrecht-Beziehung gelten kann:
 $(SKART_kn, ado_kn) \in \rho_{(v_)}^{potacc}$

Definition 3.15

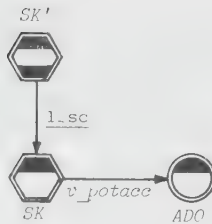
Ein Systemgraph, in dem $KB_KONS_POTZUGR$ gilt, heißt *ZR-konsistent*.

Satz 3.16

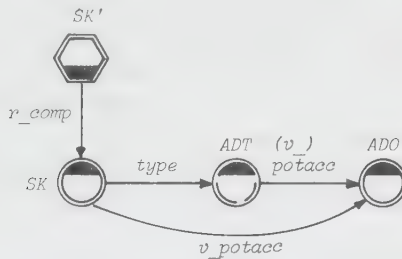
Ein Systemgraph SG ist ZR-konsistent, wenn er bzgl. der expliziten Zugriffs- und Benutzungsrecht-Beziehung konsistent ist.

Beweis

Es ist $KB_KONS_POTZUGR$ nur für den Fall zu zeigen, daß zwischen SK und ADO eine virtuelle Zugriffsrecht-Beziehung gilt. Angenommen, $KB_KONS_POTZUGR$ gilt nicht, d.h. ADO und SK besitzen keinen gemeinsamen Vater und der Vater von SK besitzt kein (explizites) Zugriffsrecht bzgl. ADO (bzw. zu wenig ZR-Attribute):



Fall 1: *SK* ist ein ADO: *SK* ist von einem ADT *ADT* und erhielt dessen explizites bzw. virtuelles Zugriffsrecht bzgl. *ADO* als virtuelles Zugriffsrecht übertragen:



a) *ADT* ist Sohn von *SK'*:

- a1) Zwischen *ADT* und *ADO* besteht explizite Zugriffsrecht-Beziehung (*potace*-Kante). *SK'* müßte wegen Konsistenz bzgl. der expliziten Zugriffsrecht-Beziehung explizites Zugriffsrecht bzgl. *ADO* (bzw. entsprechend viele ZR-Attribute) besitzen. Widerspruch.
- a2) Zwischen *ADT* und *ADO* besteht virtuelle Zugriffsrecht-Beziehung (*v_potace*-Kante). Diese Situation wird in Fall 2 abgehandelt.

b) Zwischen *SK'* und *ADT* besteht Zugriffsrecht-Beziehung: Dies widerspricht KB VERERBVERTR POTBEN, da nach Annahme *SK'* kein Zugriffsrecht bzgl. *ADO* (bzw. entsprechend viele ZR-Attribute) besitzt.

Fall 2: *SK* ist ein ADT: analog Fall 1 (Situation a2) wird in Fall 3 fortgeführt).

Fall 3: *SK* ist ein GADT: analog Fall 1 (Situation a2) wird in Fall 3 fortgeführt: Terminierung aufgrund der Endlichkeit der *g_type**-Pfade gesichert)

Fall 4: *SK* ist eine FK. Diese FK ist wegen der *v_potacc*-Kante das Ergebnis der Benutzungsspezifikation bzgl. einer PFK. Ist diese PFK Bruder zu *SK*, ergibt sich ein Widerspruch zur Vererbungsregel; wird PFK von außerhalb des *SK'*-Teilsystems benutzt, widerspricht dies *KB_VERERBVERTR_POTBEN*.

3.3.5 WEITERE KONSISTENZBEDINGUNGEN

Während die Konsistenzbedingungen in Abschnitt 3.3.3 die Korrektheit der expliziten Benutzungsrecht-Beziehung sicherstellten, müssen noch Konsistenzbedingungen für ein korrektes Ausnutzen eines Benutzungsrechts, die aktuelle Benutzung eines (G)ADT's oder einer PFK also, formuliert werden:¹⁾ Eine Systemkomponente *SK* darf von einer Systemkomponente *SK* nur dann (in der äußeren Schnittstelle oder den Realisierungs-Varianten) aktuell benutzt werden, wenn *SK* auch ein entsprechendes (explizites, implizites oder formales) Benutzungsrecht besitzt.

Ein Systemgraph, in dem die erwähnten Konsistenzbedingungen gelten, heißt *bzgl. aktueller Benutzungen konsistent*.

Definition 3.17

Ein Systemgraph heißt *konsistent*, wenn er

- a) hierarchisch,
 - b) bzgl. der expliziten Zugriffsrecht-Beziehung konsistent,
 - c) bzgl. der expliziten Benutzungsrecht-Beziehung konsistent,
 - d) bzgl. aktueller Benutzungen konsistent
- ist.

¹⁾ Auf die Korrektheit aktueller Zugriffe wird später eingegangen.

KAPITEL 4: ERSTELLUNG DER SYSTEMSPEZIFIKATION

4.1 RECHNERUNTERSTÜTZUNG BEI DER ERSTELLUNG DER SPEZIFIKATION

In Kapitel 2 wurden "wohlstrukturierte" Softwaresysteme beschrieben. In diesem Kapitel wird skizziert, inwieweit ein Software-Arbeitsplatz die Erstellung der Spezifikation solcher "wohlstrukturierten" Softwaresysteme unterstützen kann.

Folgende Anforderungen sind an einen Software-Arbeitsplatz in diesem Zusammenhang mindestens zu stellen:

- Es sind eine top-down- und bottom-up-Entwurfsstrategie, aber auch eine Entwurfsstrategie mit wechselnden Entwurfsrichtungen zu unterstützen.
- Die Systemspezifikation soll batchorientiert (also in Form größerer zusammenhängender Texteinheiten) und interaktiv (inkrementell und syntaxgesteuert) eingebracht und modifiziert werden können, wobei
 - der Entwickler Vorgaben und Zusatzinformationen erhält,
 - die Systemspezifikation zwischen Modifikationen stets konsistent ist,
 - Vollständigkeitsprüfungen vorgenommen werden.
- Es sollen mehrere Mitglieder eines Entwicklungsteams gleichzeitig an der Spezifikation arbeiten können.

4.2 BATCHORIENTIERTE ERSTELLUNG

Der vorgestellte Software-Arbeitsplatz gestattet es zum einen, die Spezifikation eines Systems über einen alphanumerischen Bildschirm *batchorientiert* einzugeben und zu ändern.¹⁾

Der Bildschirm ist dazu in zwei Bereiche unterteilt (Abb. 4-1): Der obere (größere) Bereich dient der Darstellung und Manipulation

¹⁾ Prinzipiell könnte man zumindest die Grobstruktur des Systems auch mit graphischen Mitteln spezifizieren.

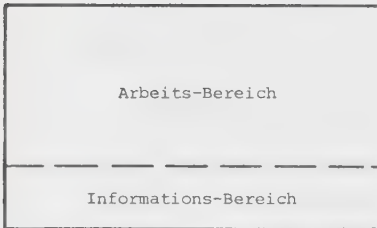


Abb. 4-1

des aktuellen Spezifikations-
texts (*Arbeits-Bereich*), der
untere der Wiedergabe von
Muenues, Vorgaben, Fehlermel-
dungen und Informationen (*In-
formations-Bereich*). Hat der
Entwickler das Eingabe-Kom-
mando abgesetzt, kann er den
Spezifikationstext im Ar-
beits-Bereich des Bildschirms
eingeben. Dazu stehen ihm

komfortable Edierfunktionen eines im Software-Arbeitsplatz inte-
grierten Texteditors zur Verfuegung. Der Eingabe-Abschluß erfolgt
entweder durch das Ignoriere- oder das Akzeptiere-Kommando. Das
Ignoriere-Kommando "vergißt" den eingegebenen Spezifikationstext;
die Eingabe ist also wirkungslos. Bei Absetzen des Akzeptiere-Kom-
mandos unterzieht ein *Spezifikations-Checker* die eingegebene Spe-
zifikation (nach der lexikalischen Analyse) einer Syntaxanalyse
(z.B. mit Hilfe des Verfahrens des rekursiven Abstiegs). Die Ana-
lyseprozeduren ueberpruefen die Konsistenz der Spezifikation hin-
sichtlich des Strukturmodells aus Abschnitt 2.2 und bauen im Falle
syntaktischer Korrektheit den Systemgraph auf.¹⁾ Bei Verstoeßen ge-
gen die (kontextfreie und kontextsensitive) Syntax der Spezifika-
tionssprache erhaelt der Systementwickler praezise Meldungen hin-
sichtlich Fehler-Art und -Ort.²⁾ Nach der Fehlerkorrektur wird die
Systemspezifikation einer neuerlichen Syntaxpruefung unterzogen.

Der waehrend der Syntaxanalyse erstellte Systemgraph wird nach der
Analyse auch im Falle einer fehlerhaften Spezifikation nicht ge-
loescht: Zur Einsparung von Speicherplatz bewahrt der Software-Ar-
beitsplatz den Quelltext der Spezifikation nicht direkt auf, son-
dern reproduziert ihn bei Bedarf durch ein *Unparse-Verfahren* aus dem
Systemgraph. Auf die dazu zusaezentlich im Systemgraph noetigen Infor-
mationen wird spaeter eingegangen.

¹⁾ Im Gegensatz zur Uebersetzung eines Programms wird jedoch aus dieser Intern-
Darstellung spaeter kein Maschinencode erzeugt.

²⁾ Es sind daefuer vielfaeltige komfortable Darstellungsmoeglichkeiten am Bildschirm
denkbar.

Bisher konnte die Spezifikation eines Systems nur in einem einzigen (monolithischen) Spezifikationstext (mit meist nicht unerheblicher Verschachtelungsstruktur) eingebracht werden. Diese Vorgehensweise führt jedoch nicht nur bald zum Verlust an Überschaubarkeit, sondern verhindert sogar ein unabhängiges Entwerfen von Teilsystemen. Gerade dies ist jedoch bei der Entwicklung eines großen Softwaresystems durch ein Entwicklungsteam unerlässlich: Jedes Teammitglied hat nach Vorgabe einer klar definierten syntaktischen und semantischen Schnittstelle einer Systemkomponente eine oder mehrere Realisierungs-Varianten bzw. ganze Teilsystem-Varianten zu entwerfen und auch dafür verantwortlich zu zeichnen. Dabei werden mehrere Teammitglieder gleichzeitig bzw. zeitlich überlappend aktiv sein. Ein physisches Zerlegen des Spezifikationstexts zur Arbeitsteilung - und damit zur Begrenzung der Verantwortlichkeit - wird durch das Konzept der *getrennten Spezifizierbarkeit* ermöglicht.

4.2.1 GETRENNTE SPEZIFIZIERBARKEIT

Das Konzept der getrennten Spezifizierbarkeit sieht vor, die Spezifikation eines Systems entweder in Form eines einzigen Textstücks oder einer Folge von Textstücken einzugeben, deren Konsistenz hinsichtlich des Strukturmodells dann *getrennt überprüft* wird. Jedes solche getrennt überprüfte Textstück selbst besteht aus einer einzigen getrennt überprüfbaren Texteinheit oder einer Folge solcher. Eine solche Texteinheit heißt *Spezifikationseinheit*.¹⁾ Im batchorientierten Eingabe- und Änderungsmodus stellt also eine Spezifikationseinheit die kleinste Texteinheit dar, die einer getrennten Konsistenzprüfung unterzogen werden kann.²⁾

Als Spezifikationseinheit ist zum einen jede Texteinheit zugelassen, die eine *Realisierungs-Variante* einer Systemkomponente spezifiziert. Diese Texteinheiten sind in sich abgeschlossene Entwurfs-

1) Diese Vorgehensweise ähnelt dem Konzept der getrennten Übersetzbarkeit von Programmeinheiten in ADA (/DoD80/).

2) Im interaktiven Entwicklungsmodus werden noch kleinere Texteinheiten Eingabe- und Änderungseinheiten bilden. Der batchorientierte und interaktive Entwicklungsmodus unterscheiden sich also u.a. in der Größe ihrer elementaren Ersetzungseinheiten.

einheiten in obigem Sinne und können relativ umfangreich sein. Zum anderen ist auch jede Texteinheit, die eine Systemkomponente komplett (also deren äußere Schnittstelle und Realisierungs-Varianten¹⁾) spezifiziert, eine Spezifikationseinheit. Diese Spezifikationseinheit nennen wir *SK-Spezifikation*. Diese zweite Möglichkeit der textuellen Auslagerung ist ebenfalls im potentiell nicht unerheblichen Textumfang begründet. Die Spezifikation lokaler Systemkomponenten könnte nämlich die umgebende Realisierungs-Variante der Vater-Komponente textuell aufblähen und unübersichtlich werden lassen.

Jede getrennt eingebrachte Spezifikationseinheit ist dort, wo sie in der Systemhierarchie einzuordnen ist, nämlich ihrer *Vater-Einheit*, durch einen *SE-Stummel* (durch separate eingeleiteter Verweis) zu verankern.

Um bei der Konsistenzprüfung einen eindeutigen Bezug zur entsprechenden Vater-Einheit zu besitzen, ist einer getrennt zu überprüfenden Spezifikationseinheit das Wortsymbol separate mit dem (i.a. qualifizierten) Namen der Vater-Einheit voranzustellen (*separate-Klausel*)²⁾. Vater-Einheit einer getrennt einzubringenden SK-Spezifikation ist eine Realisierungs-Variante³⁾ bzw. umgekehrt.

1) Sofern diese nicht wiederum getrennt spezifiziert werden.

2) Für die Konsistenzprüfung einer Spezifikationseinheit ist der Ort ihrer Verankerung relevant.

3) Ausgenommen die Systemkomponente an der Systemwurzel.

b)

```

ado ASSMEM
:
:
ado-realization REALIS1
:
:
representation
ado NM : separate;
:
end ado-realization REALIS1

end ado ASSMEM
    
```

Spezifikations-
einheit ASSMEM

```

separate ASSMEM(REALIS1)1)
ado NM
:
external interface
:
end external interface
ado-realization B_TREE_VT : separate;
:
ado-realization TABLE_VT
:
end ado-realization TABLE_VT

end ado NM
    
```

Spezifikations-
einheit NM
(getrennt spezi-
fizierte System-
komponente)

```

separate ASSMEM(REALIS1).NM
ado-realization B_TREE_VT
:
end ado-realization B_TREE_VT
    
```

Spezifikations-
einheit B_TREE_VT
(getrennt spezi-
fizierte Realisie-
rungs-Variante)

Abb. 4-2

¹⁾ Im qualifizierten Namen der übergeordneten Spezifikationseinheit sind auch die jeweils gewählten Realisierungs-Varianten anzugeben. Lokale Systemkomponenten in verschiedenen Realisierungs-Varianten dürfen nämlich gleichbezeichnet sein. Der qualifizierte Name endet mit dem Bezeichner der übergeordneten Spezifikationseinheit, gleichgültig, ob diese eine Systemkomponente oder eine Realisierungs-Variante spezifiziert.

Die Aufteilung der gesamten Systemspezifikation in einzelne Spezifikationseinheiten wird im Systemgraph folgendermaßen wiedergegeben (vgl. Beispiel Abb. 4-2 b)):

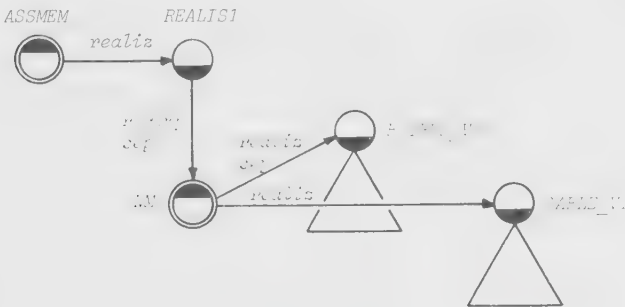


Abb. 4-3

Der Markierungs-Attributwert *sep* einer Kante gibt an, daß der am Zielknoten dieser Kante hängende Untergraph der Realisiert-Hierarchie eine getrennt eingebrachte Spezifikationseinheit repräsentiert. Dieser Untergraph ist "nach unten" durch die Kanten begrenzt, die als nächste (in der Realisiert-Hierarchie abwärts) mit dem Attributwert *sep* versehen sind.

Getrennte Konsistenzprüfung von Spezifikationseinheiten bedeutet nicht, daß diese Spezifikationseinheiten in beliebiger Reihenfolge auf Konsistenz überprüft werden können. Für eine Spezifikationseinheit ist neben ihrer "lokalen" Konsistenz auch die Korrektheit der in ihr auftretenden Bezüge zu anderen Spezifikationseinheiten zu überprüfen. Diese Außenbezüge sollten bereits bei der Erstüberprüfung vollständig überprüfbar sein, so daß eine Spezifikationseinheit zu einem späteren Zeitpunkt keinen nachträglichen ergänzenden Überprüfungen zu unterziehen ist. Die Spezifikationseinheit selbst braucht dabei nur insofern vollständig sein, als ihre "lokale" Konsistenz gesichert ist. Ein stufenweiser Ausbau einer Spezifikationseinheit bleibt also möglich.

Die Forderung nach vollständiger Überprüfung führt sofort zu einer Einschränkung hinsichtlich der Reihenfolge, in der Spezifikationseinheiten überprüft werden können (*Konsistenzprüfungs-Reihenfolge*): Eine Spezifikationseinheit mit angewandten Auftreten von Ressourcen,

die in anderen Spezifikationseinheiten definiert werden, kann erst nach diesen Spezifikationseinheiten geprüft werden.¹⁾

Mit den Regeln für die Zugriffsrecht- und Benutzungsrecht-Vergabe ergibt sich somit:

- (1) Eine Realisierungs-Variante zu einer Systemkomponente *SK* kann erst nach der Spezifikation von *SK* geprüft werden, falls in ihr Ressourcen aktuell verwendet werden, die von *SK* importiert werden.
Grund: Wegen der geforderten Vollständigkeit der Überprüfung sind erst die entsprechenden Zugriffs- und Benutzungsrechte abzuarbeiten.
- (2) Die Spezifikation einer Systemkomponente *SK* kann erst überprüft werden, wenn die Deklarationen aller ZR- bzw. BR-Objekte von *SK* bzw. (im Falle ererbter Zugriffsrechte) die Vater-Komponente von *SK* abgearbeitet sind.
Grund: Die Korrektheit eines Zugriffs- bzw. Benutzungsrechts kann (wieder wegen der Vollständigkeit der Überprüfung) erst zu diesem Zeitpunkt geprüft werden.²⁾

Die Reihenfolge-Restriktion legt auf den Spezifikationseinheiten eine *Halbordnung* fest. Es ist jede Konsistenzprüfungs-Reihenfolge zulässig, die diese Halbordnung nicht verletzt.

Beispiel:

Die Reihenfolge-Restriktion impliziert auf den Spezifikationseinheiten aus Beispiel Abb. 4-2 b)³⁾ folgende Halbordnung ("SE1 → SE2" gdw. "SE1 ist vor SE2 zu prüfen").

ASSMEM (SK-Spezifikation)

NM (SK-Spezifikation)  *B_TREE_VT* (Realisierungs-Variante)
TABLE_VT (Realisierungs-Variante)

-
- 1) In ADA bestimmt dieselbe Restriktion, bezogen auf Übersetzungseinheiten, die Übersetzungs-Reihenfolge bei getrennter Übersetzung (siehe z.B. /NAG82/).
 - 2) Da die Zugriffsrecht- bzw. Benutzungsrecht-Beziehung zyklensfrei ist, können in der Überprüfungs-Reihenfolge keine "Deadlocks" auftreten.
 - 3) Die NM-Realisierungs-Variante *TABLE_VT* sei jetzt auch getrennt spezifiziert. In *B_TREE_VT* und *TABLE_VT* treten von NM importierte Ressourcen angewandt auf. Die Spezifikationseinheiten mit den Definitionen dieser Ressourcen seien bereits geprüft.

B TREE VT kann also erst nach *NM* eingebracht werden, während für *ASSMEM* und *NM* die gegenseitige Reihenfolge beliebig ist. Verträglich mit der vorliegenden Halbordnung sind folgende Konsistenzprüfungs-Reihenfolgen (die Spezifikationseinheiten seien durch den Anfangsbuchstaben ihrer Namen gekennzeichnet):

A N B T	N A B T	N B A T	N B T A
A N T B	N A T B	N T A B	N T B A .

Der Komposition der Spezifikation des Gesamtsystems aus einzelnen Spezifikationseinheiten entspricht im Systemgraph die Komposition der Untergraphen, die die Spezifikationseinheiten repräsentieren.

Der Untergraph zu einer Spezifikationseinheit wird sofort in den Graph zur entsprechenden Vater-Einheit integriert, sobald die Beziehung Vater-Einheit $\rightarrow$ Sohn-Einheit hergestellt ist:

- Ist bei der Abarbeitung einer Spezifikationseinheit *SE* mit einer separate-Klausel die zugehörige Vater-Einheit *SE'* bereits eingebracht, wird der Untergraph zu *SE* in den für *SE'* integriert.

Existiert die Vater-Einheit *SE'* noch nicht bzw. ist *SE* keine separate-Klausel vorangestellt, wird der Untergraph zu *SE* an den mit SYSTEM markierten Systemgraph-Knoten angehängt. Dieser Knoten faßt also die Untergraphen des Hauptsystems bzw. aller Teilsysteme zusammen, die noch nicht in das Haupt- oder ein anderes Teilsystem integriert sind:

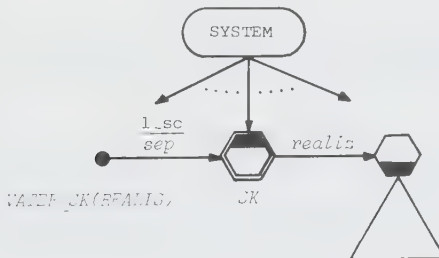


Abb. 4-4

In Abb. 4-4 ist die Spezifikationseinheit für eine Systemkomponente *SK* eingebracht. Sie weist in einer *separate*-Klausel die (noch nicht eingebrachte) Realisierungs-Variante *VATER_SK* (*REALIS*) als ihre Vater-Einheit aus. Diese Realisierungs-Variante wird durch einen abgemagerten Knoten (●) vorgemerkt.

- Bei der Abarbeitung eines SE-Stummels in einer Spezifikationseinheit *SE* (mit Bezug auf eine Spezifikationseinheit *SE'*) wird der Untergraph zu *SE'* in den für *SE* integriert, falls *SE'* bereits eingebracht ist und *SE'* keine *separate*-Klausel vorangestellt ist bzw. eine *separate*-Klausel vorangestellt ist, die *SE* als Vater-Einheit ausweist. Ist *SE'* noch nicht eingebracht, hat die Abarbeitung keinen Effekt.

Beispiel:

Es wird aus den für das Beispiel in Abb. 4-2 b) möglichen Konsistenzprüfungs-Reihenfolgen die Reihenfolge N T A B betrachtet. Zunächst wird die Spezifikationseinheit *NM* (mit der *separate*-Klausel "separate *ASSMEM* (*REALIS1*)") eingebracht:

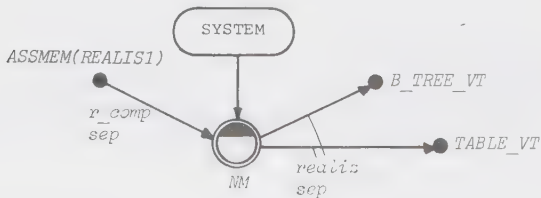


Abb.: 4-5 a

Die anschließend eingebrachte Spezifikationseinheit *TABLE VT* (mit der *separate*-Klausel "separate *NM*") wird in die Spezifikationseinheit *NM* integriert:

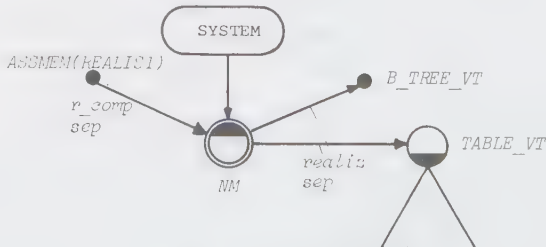


Abb. 4-5 b

Als nächste Spezifikationseinheit wird *ASSMEM* überprüft:

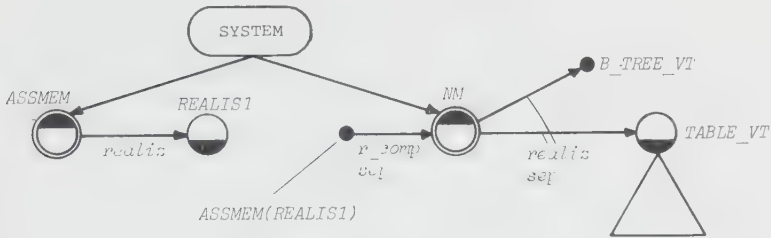


Abb. 4-5 c

Bei Abarbeitung des SE-Stummels mit Bezug auf *NM* wird das inzwischen aufgebaute Teilsystem zu *NM* in der Realisierungs-Variante *REALIS1* von *ASSMEM* integriert:

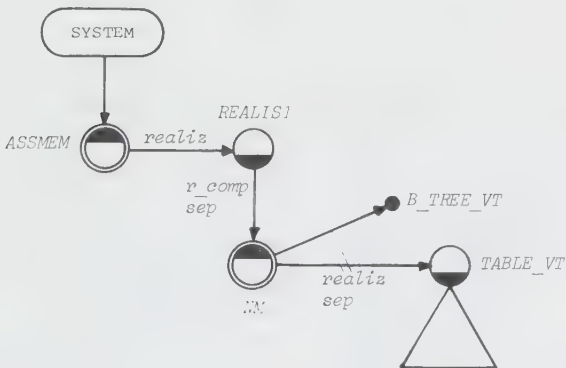


Abb. 4-5 d

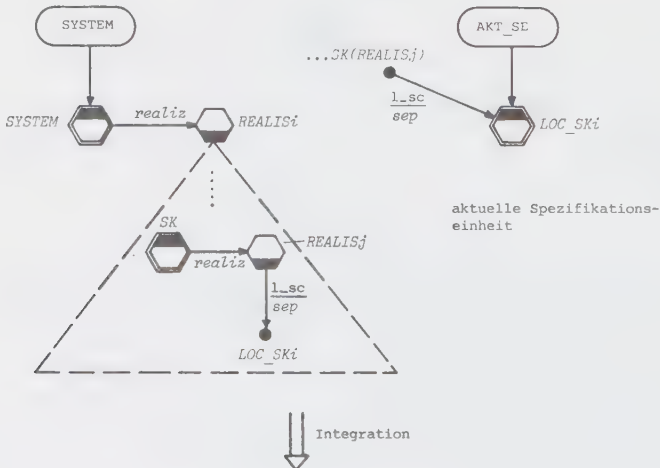
Zuletzt wird zu *NM* die Realisierungs-Variante *B_TREE_VT* eingebracht. Anschließend liegt die Situation aus Abb. 4-3 vor.

4.2.2 ENTWURFSSTRATEGIEN

Der zu konzipierende Software-Arbeitsplatz soll sowohl einen *top-down*- als auch einen *bottom-up-Systementwurf*, aber auch einen Entwurf mit wechselnden Entwurfsrichtungen ermöglichen. Im folgenden wird aufgezeigt, daß das Konzept der getrennten Spezifizierbarkeit diese verschiedenartigen Entwurfsstrategien unterstützt.

Bei einer *top-down*-Entwicklung wird jeweils die Spezifikation einer Systemkomponente SK (genauer: deren äußere Schnittstelle und Realisierung) vor der Spezifikation ihrer lokalen Systemkomponenten eingebracht. Die dadurch implizierte Reihenfolge für die Eingabe der Spezifikationseinheiten ist verträglich mit den einschränkenden Bedingungen für die Konsistenzprüfungs-Reihenfolge: ZR- und BR-Objekte einer aktuell zu spezifizierenden lokalen Systemkomponente LOC_SK_i sind aufgrund des Entwurfsprinzips des information hidings stets in höheren Hierarchieebenen angesiedelt (und deshalb bereits spezifiziert) oder liegen als Bruder-Komponenten in derselben Hierarchieebene. Innerhalb dieser Hierarchieebene dürfen jedoch keine Zugriffszyklen auftreten. Somit kann die Spezifikation von LOC_SK_i eingebracht werden, sobald sämtliche ZR- bzw. BR-Objekte derselben Hierarchieebene spezifiziert sind.

Im Systemgraph:



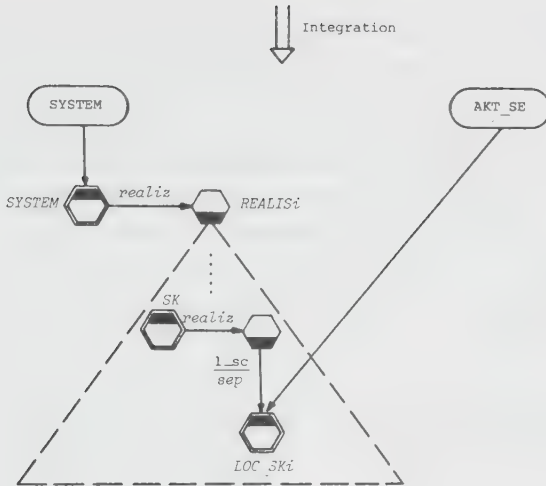


Abb. 4-6

Es ist bereits eine Systemkomponente SK mit einer Realisierungs-Variante $REALIS_j$ spezifiziert. In $REALIS_j$ verweist ein SE-Stummel auf die noch (getrennt) zu spezifizierende lokale Systemkomponente LOC_SK_i . SK ist selbst lokale Systemkomponente zur Realisierungs-Variante $REALIS_j$ des Gesamtsystems $SYSTEM$. Nun wird LOC_SK_i als eigene Spezifikationseinheit eingebracht,¹⁾ wobei die vorangestellte separate-Klausel "separate SYSTEM ($REALIS_j$) $SK(REALIS_j)$ " LOC_SK_i als die Systemkomponente ausweist, deren Verankerung im Gesamtsystem bereits durch den SE-Stummel in der SK -Realisierungs-Variante $REALIS_j$ festgelegt ist. LOC_SK_i wird deshalb sofort dort integriert.

Im Zuge einer *bottom-up*-Entwicklung ist es erforderlich, *Teilsysteme* spezifizieren zu können, deren Integration in ein anderes Teilsystem erst zu einem späteren Zeitpunkt erfolgt.

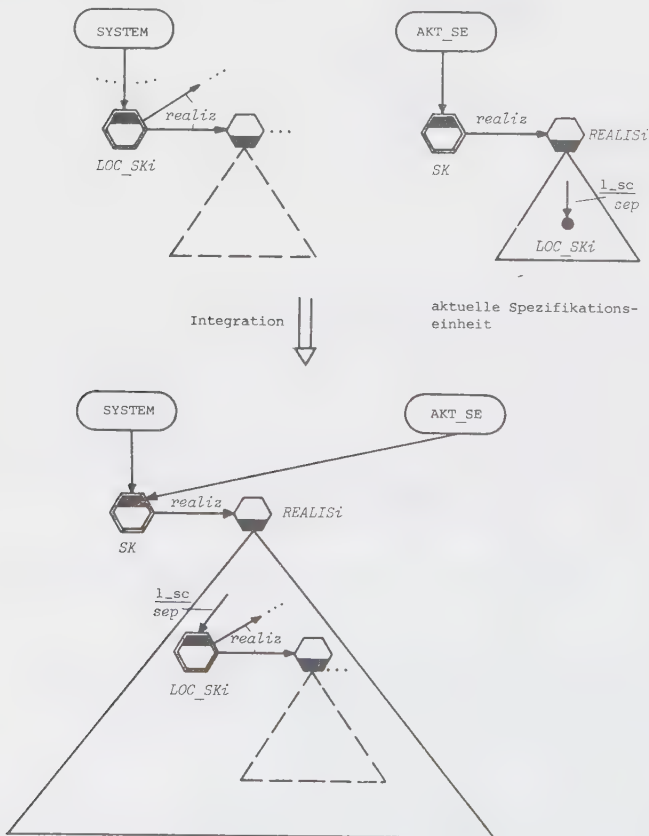
Zunächst wird also beispielsweise eine Systemkomponente LOC_SK_i spezifiziert, deren Verankerung im (noch zu vervollständigenden) Gesamtsystem noch nicht festliegt.²⁾

¹⁾ Der AKT_SE -Knoten hält stets den Untergraph der gerade überprüften Spezifikationseinheit (aktuelle Spezifikationseinheit) fest.

²⁾ Die entsprechende Spezifikationseinheit besitzt keine separate-Klausel.

Später kann das durch LOC_SKi aufgespannte Teilsystem beispielsweise in einer Teilsystem-Variante einer anderen Systemkomponente SK integriert werden (Abb. 4-7). Dabei wird SK (mit Realisierungs-Variante $REALISi$) nach LOC_SKi spezifiziert und spannt ein weiteres Teilsystem oder das Hauptsystem auf. Die Verankerungsstelle für LOC_SKi ist durch einen entsprechenden SE-Stummel fixiert. Die Integration wird vollzogen, wenn während der Überprüfung von $SK(REALISi)$ dieser SE-Stummel abgearbeitet wird.

Im Systemgraph:



Es erhebt sich nun die Frage, welche Bezüge zwischen verschiedenen, noch nicht integrierten Teilsystemen zuzulassen sind.

Bei einer (reinen) bottom-up-Entwicklung werden sämtliche Sohn-Systemkomponenten vor ihrer Vater-Komponente spezifiziert. Die Zugriffsrecht-/Benutzungsrecht-Beziehungen von Abb. 4-8a zwischen zwei Teilsystemen zu Systemkomponenten *SK1* und *SK2* erweisen sich deshalb als korrekt, wenn *SK1* und *SK2* zu einem späteren Zeitpunkt als Bruder-Komponenten in eine gemeinsame Vater-Komponente integriert werden¹⁾ (im restlichen Abschnitt werden Systemgraph-Ausschnitte nur schematisiert gezeichnet). In der Entwurfsphase dürfen jedoch zwischenzeitlich auch die Zugriffsrecht- bzw. Benutzungsrecht-Beziehungen von Abb. 4-8b auftreten. Diese Bezüge erweisen sich als verträglich mit dem Strukturmodell, falls das Teilsystem *SK1* im Teilsystem zu *SK2* auf gleicher oder niedrigerer Hierarchieebene wie *SK4* integriert wird.

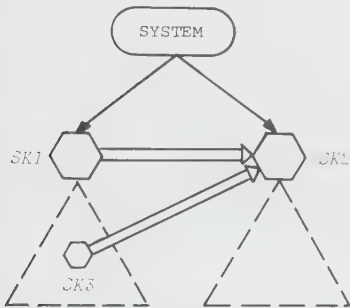


Abb. 4-8 a

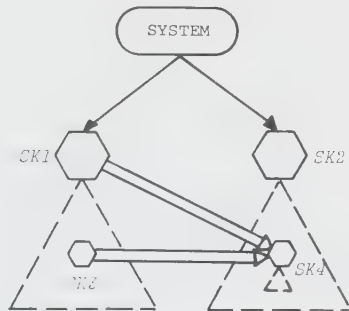


Abb. 4-8 b

Prinzipiell sind also zwischen noch nicht integrierten Teilsystemen zunächst sämtliche denkbaren Zugriffsrecht-/Benutzungsrecht-Beziehungen zuzulassen (Abb. 4-8c), da die Korrektheit dieser Beziehungen sowieso erst zum Zeitpunkt der Integration überprüft werden

¹⁾ Bei einer reinen top-down-Entwicklung werden neu eingebrachte Teilsysteme sofort in das Hauptsystem integriert. Somit treten keine Bezüge zwischen noch nicht integrierten Teilsystemen auf.

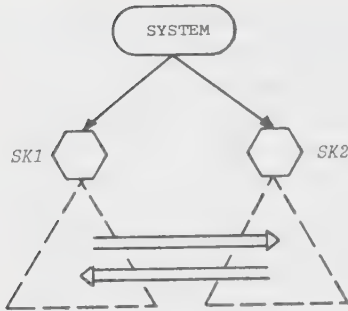


Abb. 4-8 c

Die oben geforderte Vollständigkeit der Überprüfung einer Spezifikationseinheit ist also insofern nicht in vollem Umfang realisierbar, als beim Eintrag eines Zugriffs- bzw. Benutzungsrechts zunächst nur die Korrektheit der ZR- bzw. BR-Attribute überprüfbar ist, falls das ZR- bzw. BR-Objekt in einem anderen noch nicht integrierten Teilsystem liegt. Bei einer Integration des Teilsystems zu einer Systemkomponente *SK1* in das zu einer Systemkomponente *SK2* ist dann nachträglich dieser Zugriffsrecht- bzw. Benutzungsrecht-Bezug daraufhin zu überprüfen, ob das ZR- bzw. BR-Objekt auch tatsächlich im Zugriffs- bzw. Benutzungsbereich des ZR- bzw. BR-Subjekts zu liegen kommt.¹⁾ Die Integration wird dann nur im Falle der Fehlerfreiheit vollzogen.

Betrachtet man Teilsystem-Beziehungen nicht-isoliert, lassen sich vor der Integration bereits Konstellationen bestimmen, die nach der Integration zwangsweise zu Inkonsistenzen führen, beispielsweise die Konstellation (/WES82/):

Derartige "Frühdiagnosen" tragen zur Erfüllung der Forderung bei, auf Verstöße gegen die Strukturkonsistenz frühestmöglich zu reagieren.

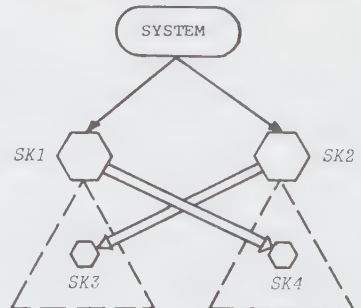


Abb. 4-8 d

¹⁾ Bezüge zu dritten Teilsystemen bedürfen keiner Überprüfung.

4.2.3 ANMERKUNGEN ZUM UNPARSE-VERFAHREN

Der vorgestellte Software-Arbeitsplatz bewahrt Spezifikationseinheiten nicht in Quelltext-Form auf, sondern legt ihre semantischen Informationen komprimiert im Systemgraph ab. Aus dem Systemgraph muß deshalb stets der momentane Bearbeitungszustand einer Systemspezifikation eindeutig rekonstruierbar sein.

Die semantischen Informationen werden im Systemgraph gemäß der Systemgraph-Definition (vgl. Abschnitt 3.2) repräsentiert. Die Reihenfolge, in der sie im Quelltext einer Spezifikationseinheit eingebracht wurden, ist in der Syntax unserer Spezifikationssprache¹⁾ fixiert, also rekonstruierbar. Eine Ausnahme bilden Listenelemente. Zur Rekonstruierbarkeit ihrer gegenseitigen Reihenfolge ist ihre Systemgraph-Darstellung noch um eine zusätzliche Knoten- und Kantenstruktur zur Reihenfolge-Organisation zu ergänzen.

Beispiele:

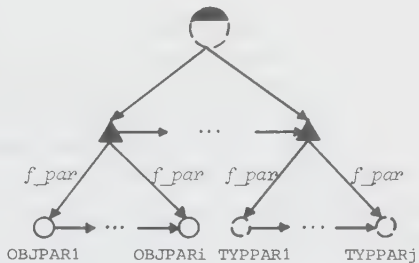
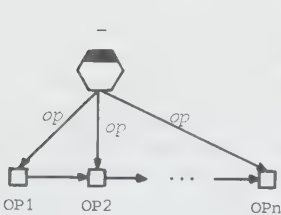


Abb. 4-9 a: Liste zu exportierender Operationen/Funktionen

Abb. 4-9 b: Liste generischer Parameter eines GADT's

Im Beispiel von Abb. 4-9b dient der Knoten ▲ (*Listenorganisator*) der Zusammenfassung mehrerer untergeordneter Listenelemente zu einem übergeordneten. Dies ist stets dann nötig, wenn die Einführung eines speziellen Knotens für die Gesamtliste in der Systemgraph-Darstellung zunächst nicht vorgesehen wurde.

¹⁾ Im Anhang ist die Syntax der vorgeschlagenen Spezifikationssprache als Zweistufen-Grammatik in der erweiterten Backus-Naur-Notation (EBNF) angegeben. Komplexere kontextsensitive Abhängigkeiten sind in Form von Bemerkungen beschrieben.

Aus Lesbarkeitsgründen wird im folgenden öfters von der listenorganisierenden Knoten-/Kantenstruktur abstrahiert werden. Durch die Notation in Abb. 4-9c wird beispielsweise die i -te Operation einer Systemkomponente, durch die Notation in Abb. 4-9d der dritte formale generische Parameter in der zweiten Formalparameter-Deklaration angesprochen.¹⁾

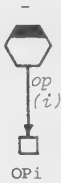


Abb. 4-9 c

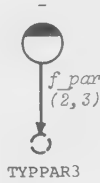


Abb. 4-9 d

Um nicht Informationen zu der vom Entwickler vorgenommenen Formatierung des Spezifikationstextes ablegen zu müssen, nimmt der Software-Arbeitsplatz bei der Ausgabe einer Spezifikationseinheit automatisch eine spezielle Formatierung vor. Dazu ist ein Formatierer nötig, der vom Entwickler parametrisierbar sein sollte.

4.3 INKREMENTELLE UND SYNTAXGESTEUERTE ERSTELLUNG

Der vorgestellte Software-Arbeitsplatz sieht neben dem batchorientierten auch einen interaktiven Entwicklungsmodus vor: Die Systemspezifikation kann inkrementell und syntaxgesteuert eingegeben bzw. geändert werden.

Waren im batchorientierten Entwicklungsmodus die Spezifikationseinheiten elementare Einheiten hinsichtlich Eingabe und Änderung, sind im *inkrementellen Entwicklungsmodus* die Ersetzungseinheiten feiner: Jede Spezifikationseinheit ist in eine Hierarchie von *Inkrementen* untergliedert, die einzeln eingegeben und geändert werden können. Die Fixierung des Inkrement-Begriffs ist dabei nicht willkürlich: Inkremente stellen gewisse syntaktische Einheiten der Spezifikationssprache dar.²⁾ Es wird sich zeigen, daß aus der Sicht des inkrementellen Entwicklungsmodus auch eine ganze Spezifika-

1) Ist im folgenden bei der Darstellung eines bestimmten Sachverhalts die konkrete Listenposition ohne Bedeutung, werden keine Reihenfolge-Angaben gemacht.

2) Der Inkrement-Begriff wird im nächsten Abschnitt noch präzisiert werden.

tionseinheit ein Inkrement, und zwar das größtmögliche Inkrement, darstellt. Das über den batchorientierten Entwicklungsmodus Gesagte bleibt deshalb auch für die inkrementelle Entwicklung gültig: Die Spezifikation kann in Form einzelner Spezifikationseinheiten eingebracht werden. Da die Überprüfung ihrer gegenseitigen Bezüge weitestgehend vollständig sein soll, unterliegt die Reihenfolge ihrer Eingabe den bekannten Konsistenzprüfungs-Regeln. Eine Spezifikationseinheit selbst braucht dabei nur insoweit vollständig sein, als zwischen Modifikationen ihre "lokale" Konsistenz stets gesichert ist. Dies bedeutet insbesondere, daß einem angewandten Auftreten einer Größe stets ihr definierendes Auftreten vorangehen muß.¹⁾ Die Darstellung einer Spezifikationseinheit am Bildschirm erfolgt wieder über ein Unparse-Verfahren.

Der vorliegende Abschnitt beschäftigt sich also nur mit Aspekten, die aus der Verfeinerung der Struktur einer Spezifikationseinheit resultieren.

Neu eingegebene bzw. geänderte Inkremente werden durch einen *inkrementellen Spezifikations-Checker* sofort auf ihre Korrektheit hinsichtlich kontextfreier und kontextsensitiver Syntax überprüft. Da sie im Fehlerfall zurückgewiesen werden, verbleibt die Spezifikation des zu entwickelnden Softwaresystems stets in einem konsistenten Zustand. Ist ein neues Inkrement syntaktisch korrekt, wird der Systemgraph durch einen entsprechenden Untergraph ergänzt.

Der Entwickler wird im inkrementellen Entwicklungsmodus durch einen *syntaxgesteuerten Spezifikations-Editor* unterstützt. Dieser Editor bietet dem Entwickler Inkremente zur Auswahl an, die an der aktuellen Eingabe-/Änderungsposition syntaktisch zulässig sind, und generiert automatisch Teile ihrer konkreten Syntax.

1) Man könnte stattdessen zur Erhöhung des Eingabe-Komforts abschwächend fordern, daß der Abgleich angewandtes $\leftrightarrow$ definierendes Auftreten spätestens bis zum Abschluß der Bearbeitung der aktuellen Spezifikationseinheit durchgeführt ist. Dieser Aspekt wird jedoch nicht weiterverfolgt.

4.3.1 BENUTZERSCHNITTSTELLE

4.3.1.1 BEGRIFF DES INKREMENTS

Der Entwickler baut die aktuelle Spezifikationseinheit inkrementell auf bzw. modifiziert sie inkrementell. Die *Inkremente* stellen dabei syntaktische Einheiten der Spezifikationssprache dar und sind allein aufgrund der Grammatik definiert, die dieser Sprache zugrundeliegt.¹⁾

Es lassen sich zunächst *atomare* und *komplexe* Inkremente unterscheiden.

Atomare Inkremente sind in keine weiteren Inkremente untergliedert. Sie stellen die syntaktischen Einheiten dar, die - abgesehen von Wort- und Trennsymbolen - durch die Terminalsymbole der kontextfreien Grammatik für die (konkrete) Syntax der vorgeschlagenen Spezifikationssprache definiert werden.²⁾ Es handelt sich dabei syntaktisch i.a. um Bezeichner oder Namen, die Elemente der aktuellen Spezifikationseinheit oder anderer Spezifikationseinheiten identifizieren.

Komplexe Inkremente sind die syntaktischen Einheiten, die durch die Nichtterminalsymbole der Grammatik definiert werden. Sie setzen sich aus weiteren komplexen oder aber atomaren Inkrementen zusammen, die in ein Rahmengerüst aus Wort- und Trennsymbolen eingebettet sind. Die Spezifikationseinheiten sind die größten komplexen Inkremente.

Beispiel:

Eine ADO-Deklaration mittels implizitem ADT ist in der vorgeschlagenen Spezifikationssprache definiert als³⁾

-
- 1) Bei einer anderen als der vorgeschlagenen Spezifikationssprache ergeben sich natürlich andere Inkremente.
 - 2) Atomare Inkremente entsprechen also den Terminalsymbolen der abstrakten Syntax.
 - 3) Unterstrichene Symbole (Wortsymbole) und 'ado_bez' sind terminal, die übrigen Symbole nichtterminal. 'sk_sep_klaus' ist wegen der Klammerung durch [] ein optionales Nichtterminalsymbol.

```
ado_dekl_m_impl_adt ::=
    [sk_sep_klaus]
    ado ado_bez
        ado_aeuss_schnittst
        ado_realisvarianten
    end ado ado_bez .
```

Das Nichtterminalsymbol 'ado_dekl_m_impl_adt' definiert ein komplexes Inkrement, das sich aus weiteren Inkrementen und einer Rahmenstruktur zusammensetzt: Das terminale Symbol 'ado_bez' für den ADO-Bezeichner repräsentiert ein atomares Inkrement, 'sk_sep_klaus' (separate Klausel bei getrennter Spezifizierung), 'ado_aeuss_schnittst' (äußere Schnittstelle) und 'ado_realisvarianten' (Realisierungs-Varianten) dagegen definieren komplexe Inkremente. Die Wortsymbole ado und end ado sind als Elemente der Rahmenstruktur keine Inkremente. Das komplexe Inkrement einer separate-Klausel ist gemäß der Grammatikregel

```
sk_sep_klaus ::=
    separate SKrealis_name; 1)
```

unterstrukturiert in die Rahmenstruktur "separate ...;" und das atomare Inkrement für den Namen einer Realisierungs-Variante. Im Gegensatz hierzu betten die komplexen Inkremente zu 'ado_aeuss_schnittst' bzw. 'ado_realisvarianten' ihrerseits wieder komplexe Inkremente ein.

Ein komplexes oder atomares Inkrement heißt *obligat* bzw. *optional*, falls bei seiner grammatikalischen Ableitung das Grammatik-Symbol, das das Inkrement definiert, zuletzt ungeklammert bzw. durch [] geklammert auftrat.

Inkremente, die Elemente einer gemeinsamen Liste darstellen und in der EBNF-Grammatik durch ein in { } eingeschlossenes (Nicht-)Terminalsymbol repräsentiert werden, heißen *repetitiv*. Jedes repetitive Inkrement ist optional.²⁾

Alternative Inkremente sind komplexe oder atomare Inkremente, die durch die alternativen rechten Seiten einer bestimmten Grammatik-Regel definiert werden. Sie können abhängig von der jeweiligen Form

1) SK ist ein Nichtterminalsymbol der Metagrammatik.

2) Eine Liste, die mindestens ein Element besitzen muß, wird in der Grammatik durch 'list ::= list_elem {,list_elem}' definiert. Das (obligate) Inkrement für das erste Listenelement ist somit nicht repetitiv.

des Auftretens der linken Seite dieser Regel obligat oder optional (evtl. sogar repetitiv) sein.

Neben der absoluten Klassifizierung in komplexe und atomare Inkremente lassen sich noch die Inkrementbeziehungen "global" und "lokal" einführen, die die Lage eines Inkrements relativ zu anderen Inkrementen charakterisieren. Ein Inkrement *i1* heißt *lokal* zu einem anderen (komplexen) Inkrement *i2*, wenn *i1* eines der Inkremente ist, aus denen sich *i2* (direkt oder indirekt) zusammensetzt. Umgekehrt heißt *i2 global* zu *i1*.

Eine Spezifikationseinheit bildet somit die Wurzel der Inkrementhierarchie, die durch die "global-lokal"-Beziehung gegeben ist.

4.3.1.2 INKREMENTOPERATIONEN

Die auf Inkrementen definierten Operationen (*Inkrementoperationen*) sind für komplexe und atomare Inkremente verschieden.

Das *Erzeugen* eines komplexen Inkrements bedeutet die automatische Generierung der aus Wort- und Trennsymbolen bestehenden Rahmenstruktur des Inkrements.¹⁾ Diese automatische Vorgabe durch den Spezifikations-Editor vermeidet Syntaxfehler, die bei einer zeichenweisen Eingabe der Rahmenstruktur durch den Entwickler entstehen könnten.

Das *Löschen* eines komplexen Inkrements bewirkt das Löschen sämtlicher seiner lokalen Inkremente und anschließend das Löschen der Rahmenstruktur.

Während die Rahmenstruktur eines komplexen Inkrements vom System vorgegeben wird, sind bei der *Eingabe* von atomaren Inkrementen Benutzereingaben erforderlich (z.B. Bezeichner eines ADO's). Dabei können Syntaxfehler auftreten (z.B. doppelte Bezeichner-Vergabe). Das atomare Inkrement wird deshalb sofort nach Abschluß seiner Eingabe durch den Spezifikations-Checker auf seine syntaktische Korrektheit hin überprüft. Der Spezifikations-Editor unterstützt dabei

¹⁾ Die Expansion hinsichtlich seiner lokalen Inkremente steht also noch aus.

die Syntaxanalyse insofern, als ihm die erwartete syntaktische Kategorie des Inkrements bekannt ist.

Das *Löschen* eines atomaren Inkrements ist nicht in jedem Fall zulässig. Beispielsweise darf der Bezeichner einer Systemkomponente oder einer Realisierungs-Variante vom Entwickler nicht gelöscht werden. Stattdessen wird er beim Löschen des direkt übergeordneten globalen Inkrements automatisch mitgelöscht.

Auch das *Ändern* atomarer Inkremente ist nur in manchen Fällen sinnvoll (z.B. Bezeichner einer Systemkomponente), während es in anderen Fällen auf Löschen und neues Eingeben zurückgespielt wird (z.B. Spezifikation eines ZR- oder BR-Objekts). Das Ändern komplexer Inkremente läßt sich aufgrund der unveränderlichen Rahmenstruktur auf das Ändern der entsprechenden lokalen atomaren Inkremente zurückführen.

Alle Arten inkrementeller Ersetzungen (Erzeugen, Eingeben, Löschen, Ändern) nennen wir zusammenfassend *Modifikationen*.

Den momentanen Bearbeitungszustand der aktuellen Spezifikationseinheit kann man sich in einem Ableitungsbaum repräsentiert denken. Dieser Ableitungsbaum basiert auf der abstrakten Syntax der Spezifikationssprache, d.h. die Knoten des Ableitungsbaums stehen nur für komplexe bzw. atomare Inkremente. Jede Modifikation eines Inkrements resultiert in einer inkrementellen Transformation der aktuellen Struktur des Ableitungsbaums in eine Folgestruktur.

4.3.1.3 SYNTAXGESTEUERTER SPEZIFIKATIONS-EDITOR

Die Inkremente der aktuellen Spezifikationseinheit werden am Bildschirm i.a. kaskadenförmig aufbereitet ausgegeben: Lokale Inkremente mit gemeinsamem globalem Inkrement sind in vertikaler Richtung aneinandergereiht, wobei sie gegenüber der Rahmenstruktur des globalen Inkrements etwas horizontal versetzt werden. Diese Baumdarstellung wird jedoch in Fällen durchbrochen, in denen sich eine andere Notation allgemein eingebürgert hat.¹⁾

¹⁾ z.B. die "horizontale" Schreibweise des Prozedurkopfs einer Operation/Funktion.

Der syntaxgesteuerte Spezifikations-Editor zeigt dem Entwickler an, an welchen Positionen der aktuellen Spezifikationseinheit noch Inkremente eingegeben werden müssen (obligate Inkremente) bzw. können (optionale Inkremente) und von welcher Art diese Inkremente sein müssen. Diese Informationen werden zur Minimierung der Blickwechsel zwischen unterem und oberem Bildschirm-Bereich direkt im Spezifikationstext angeboten.

Noch nicht erzeugte komplexe Inkremente bzw. noch nicht eingegebene atomare Inkremente werden durch einen *Platzhalter* in Form eines kurzen Kommentars angedeutet.¹⁾ Diese Platzhalter sollten optisch von den Rahmenstrukturen bzw. den atomaren Inkrementen abgehoben werden (z.B. durch Schwarz-Weiß-Invertierung oder bei einem Farb-Bildschirm durch entsprechende Farbwahl). Ebenso sollten sich die Platzhalter obligater von denen optionaler Inkremente unterscheiden.

Repetitive, noch nicht eingebrachte Inkremente einer Liste werden zusammenfassend durch einen '*' als Platzhalter (*-*Platzhalter*) repräsentiert. Er ermöglicht als "Listenelement-Generator" das Erzeugen/Eingeben beliebig vieler repetitiver Inkremente am Listende. Aus Lesbarkeitsgründen wird ein *-Platzhalter jedoch nur ausgegeben, solange die zugehörige Liste bearbeitet wird. Ebenfalls aus Lesbarkeitsgründen wird nicht jede Listenposition, die für die Eingabe eines zusätzlichen repetitiven Inkrements in Frage kommt, durch einen Platzhalter markiert.²⁾

Beispiel:

Das (noch nicht vollständig expandierte) komplexe Inkrement für die äußere Schnittstelle eines ADO's wird folgendermaßen dargestellt:

1) Die direkte Verwendung der Grammatiksymbole als Platzhalter wäre nicht sehr benutzerfreundlich.

2) Ausnahme: das erste Element einer Liste, die auch leer sein darf.

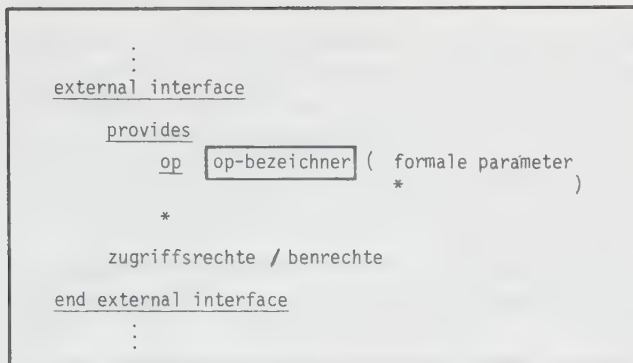


Abb. 4-10

Das obligate atomare Inkrement¹⁾ für den Bezeichner der ersten ADO-Operation ist noch nicht eingebracht. Der Platzhalter 'formale parameter' deutet das optionale komplexe Inkrement für die erste Parameterdeklaration an. Die beiden *-Platzhalter repräsentieren weitere Parameterdeklarationen bzw. weitere ADO-Operationen. 'zugriffsrechte / benrechte' ist der Platzhalter für das optionale komplexe Inkrement zur Spezifikation der Außenbezüge.

Während der interaktiven Erstellung der Spezifikation markiert der sog. *EDIT-Zeiger* (Bildschirm-Cursor ►) das *aktuelle Inkrement* (bzw. den *aktuellen Platzhalter*) in der aktuellen Spezifikationseinheit.

Der EDIT-Zeiger wird einerseits nach der Modifikation des aktuellen Inkrements/Platzhalters vom System automatisch vor das Inkrement weiterversetzt, das am wahrscheinlichsten Kandidat für die folgende Modifikation ist.²⁾

Andererseits kann der Entwickler den EDIT-Zeiger über eigene *Zeiger-Kommandos* syntaxgesteuert vor jedes beliebige Inkrement bzw. jeden beliebigen Platzhalter der aktuellen Spezifikationseinheit positionieren. Diese Kommandos sind durch die Funktionstasten ↑, ↓, ← und → für einen Einzelschritt der Aufwärts-/Abwärtsbewegung bzw. der Bewegung nach links/rechts realisiert. Die Kommandos wir-

1) Obligate atomare Inkremente werden durch die Umrahmung charakterisiert.

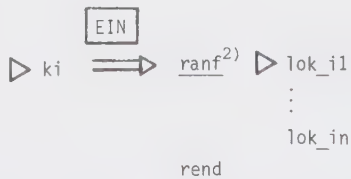
2) Dieser Kandidat ist bestimmt durch die Reihenfolge, die eine pre-order-Traversierung des aktuellen Ableitungsbaums auf den Baum-Knoten impliziert.

ken relativ zur aktuellen Position in der Bildschirm-Ebene. Zielposition ist stets wieder ein Inkrement/Platzhalter. Die Kommandos entsprechen i.a. im Ableitungsbaum (in obiger Reihenfolge) dem Übergang zum linken/rechten Bruder-Inkrement bzw. zum Vater- bzw. ersten Sohn-Inkrement. Ausnahmen sind durch komplexe Inkremente gegeben, die am Bildschirm nicht in Baumform dargestellt werden. Für die Bearbeitung dieser Inkremente sollten jedoch die Funktionstasten aus Transparenzgründen weiterhin direkt die entsprechende Vertikal- bzw. Horizontal-Bewegung in der Bildschirm-Ebene realisieren.

Sind die Funktionstasten mit einer Wiederholungsfunktion ausgestattet, läßt sich die Positionierungszeit optimieren. Ein automatischer Rolling-Mechanismus verhindert, daß der EDIT-Zeiger den Informations-Bereich des Bildschirms verläßt.

Es werden nun die Kommandos zur Realisierung der Inkrement-Operationen (*Inkrement-Kommandos*) diskutiert.

Ist der EDIT-Zeiger vor dem Platzhalter 'ki' für ein komplexes Inkrement positioniert, wird bei Drücken der Funktionstaste¹⁾ EIN das entsprechende Inkrement *erzeugt*. Dazu wird anstelle 'ki' die Rahmenstruktur des Inkrements und die Platzhalter der lokalen Inkremente ausgegeben. Der EDIT-Zeiger deutet anschließend auf den Platzhalter des ersten lokalen (komplexen oder atomaren) Inkrements 'lok_i1':



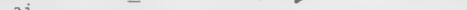
Zur *Eingabe* eines atomaren Inkrements: Bereits bei der Positionierung des EDIT-Zeigers vor die Eingabestelle wird automatisch dadurch Platz für die einzugebende Zeichenfolge geschaffen, daß

-
- 1) Inkrement-Kommandos sollten zur Vermeidung von Schreibfehlern und zur Verkürzung der Eingabezeit über spezielle Funktionstasten realisiert werden.
- 2) ranf und rend symbolisieren Beginn und Ende der Rahmenstruktur.

der Platzhalter 'ai' für das Inkrement in die nächste Zeile rutscht. Dazu wird evtl. eine Freizeile (durch Verschieben des gesamten folgenden Spezifikationstexts nach unten) geschaffen:

$\triangleright$ -Posit.
 $\dots, a_i, \dots \Rightarrow \dots, \triangleright \blacksquare, \dots$
 a_i

Nun kann das atomare Inkrement ab der Position, die durch den Eingabe-Cursor ■ markiert ist, zeichenweise eingegeben werden. Ist die Lücke bis zum nächsten Wort-/Trennsymbol zu klein, wird der Zeilenrest automatisch zeichenweise nach rechts verschoben. Ein Drücken der Return-Taste RET schließt die Eingabe ab.¹⁾ Evtl. noch vorhandener Leerraum (das Inkrement ist kürzer als sein Platzhalter) wird dabei durch automatisches Verschieben der Restzeile nach links getilgt:



Der Platzhalter 'ai' verschwindet. Der EDIT-Zeiger wird automatisch zum Platzhalter 'br_i' des nächsten (komplexen oder atomaren) Bruder-Inkrementes versetzt.²⁾ Existiert kein solcher Platzhalter mehr, wird der EDIT-Zeiger auf den (im Sinne einer pre-order-Traversierung des Ableitungsbaums) nächsten Platzhalter (bzw. Inkrement) positioniert.

Bisher wurden nur komplexe Inkremente mit ausschließlich obligaten bzw. nichtrepetitiven optionalen lokalen Inkrementen betrachtet. Besitzt ein komplexes Inkrement auch repetitive lokale Inkremente, wird der Platzhalter 'lok_i' für das erste (komplexe oder atomare) Inkrement der Liste³⁾ ausgegeben:

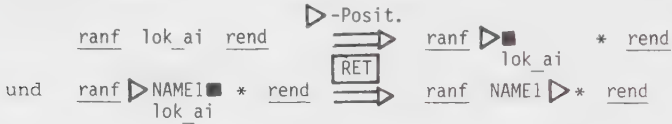
```
... ranf lok i rend
```

1) Ein Zeiger-Kommando anstelle **RET** bewirkt vor der EDIT-Zeiger-Positionierung implizit einen Eingabe-Abschluß.

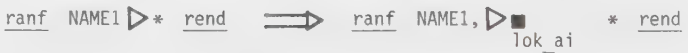
2) Anstelle des Platzhalters könnte natürlich auch bereits das Bruder-Inkrement selbst existieren.

3) Dieses Inkrement kann obligat (Liste mit mindestens einem Element) oder optional sein.

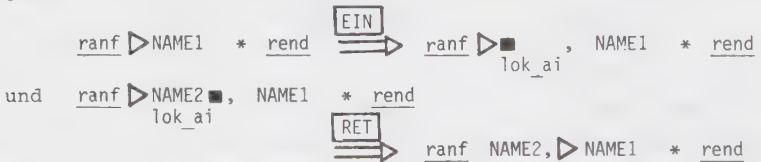
Die Eingabe des ersten Listenelements vollzieht sich wie im Nicht-listen-Fall^{1) 2)}:



Soll als zweites Listenelement NAME2 hinter NAME1 eingegeben werden, wird durch **EIN** ein Trennsymbol ausgegeben und Platz zur Eingabe reserviert:



Der eigentliche Eingabeprozess läuft wie oben beschrieben ab. Bei der Eingabe von NAME2 vor NAME1 ist zunächst der EDIT-Zeiger auf NAME1 einzustellen. Dann erfolgt die Platzreservierung und Eingabe:



Die Eingabe eines neuen Listenelements erfolgt also prinzipiell vor dem aktuell markierten Listenelement bzw. dem *-Platzhalter am Listeneende (die folgenden Listenelemente werden zur Platzreservierung entsprechend verschoben).

Zur Beschleunigung des Eingabeprozesses könnten prinzipiell auch mehrere atomare Inkremente auf einmal als eine Zeichenkette (mit entsprechenden Trennsymbolen) eingebracht werden.

Sind die Listenelemente komplexe Inkremente, bewirkt das Erzeugen eines neuen Listenelements den Einschub seiner Rahmenstruktur vor dem aktuellen Listenelement.³⁾

- 1) Im folgenden werden die Listenelemente als atomar angenommen.
- 2) Der zusätzlich ausgegebene *-Platzhalter spielt die Rolle des Bruder-Inkrementen bzw. dessen Platzhalters.
- 3) Steht der EDIT-Zeiger vor dem Platzhalter für ein Listenelement (beispielsweise für das erste Listenelement), wird durch **EIN** dieser Platzhalter expandiert.

Komplexe Listenelemente dürfen selbst wieder Listenstruktur besitzen.

Wird die Expansion eines Platzhalters für mehrere alternative Inkremente angestoßen, hat der Entwickler anhand eines Menues, das ihm im Informations-Bereich des Bildschirms angeboten wird, eine dieser Alternativen auszuwählen.

Allgemein wird die Effizienz des Eingabeprozesses gesteigert, falls Platzhalter für obligate lokale komplexe Inkremente ihrerseits sofort automatisch expandiert werden.¹⁾

Beim *Löschen* eines optionalen komplexen Inkrements (durch Betätigen der Funktionstaste LOE) wird die gesamte Hierarchie seiner lokalen Inkremente und seine Rahmenstruktur getilgt. Es erscheint sein Platzhalter 'ki' ²⁾:



Der EDIT-Zeiger wird anschließend vor den Platzhalter 'br_i' des nächsten Bruder-Inkrements (bzw. das Bruder-Inkrement selbst) positioniert. War das gelöschte Inkrement das letzte in einer Reihe lokaler Inkremente, wird der Nachfolger-Platzhalter bzw. das Nachfolge-Inkrement des zugehörigen globalen Inkrements markiert. Ist das zu löschende komplexe Inkrement obligat, wird nach dem Löschen sofort wieder dessen Rahmenstruktur ausgegeben.

Analoges gilt für das Löschen eines atomaren Inkrements.

Das *Ändern* des aktuell markierten atomaren Inkrements wird über die Funktionstaste AEN angestoßen. Dabei wandert die zu ändernde Zeichenkette in eine unterhalb des Inkrements eingeschobene Freizeile. Dann wird die neue Zeichenkette vom EDIT-Zeiger ab eingege-

1) Falls dieser Platzhalter natürlich nicht alternative Inkremente vertritt.

2) War das gelöschte Inkrement Element einer Liste, wird die Ausgabe des Platzhalters unterdrückt.

ben. Bei entsprechendem Platzbedarf wird dabei der Inhalt der Restzeile zeichenweise nach rechts verschoben. Nach Abschluß der Änderung wird andererseits bei kürzerem neuen Inkrement der Restzeilen-Inhalt linksbündig angeschlossen:

$$\begin{array}{ccc} \dots, \triangleright \text{NAMEALT}, \text{br}_i, \dots & \xrightarrow{\text{AEN}} & \dots, \triangleright \blacksquare, \text{br}_i, \dots \\ & \xrightarrow{\text{RET}} & \text{NAMEALT} \\ \text{und } \dots, \triangleright \text{NNEU} \blacksquare, \text{br}_i, \dots & \xrightarrow{\quad} & \dots, \text{NNEU}, \triangleright \text{br}_i, \dots \end{array}$$

Prinzipiell ließe sich auch die Änderung mehrerer atomarer Inkremente in einem Änderungsprozeß vorsehen.

Die Bearbeitung der aktuellen Spezifikationseinheit wird durch ein spezielles Eingabe- bzw. Änderungs-Kommando eingeleitet. Der Bearbeitungs-Abschluß erfolgt - wie bei der batchorientierten Bearbeitung - durch ein Ignoriere- oder Akzeptiere-Kommando. Das Ignoriere-Kommando "vergißt" die Modifikationen der letzten Bearbeitung, während sie durch das Akzeptiere-Kommando festgeschrieben werden.

4.3.2 IMPLEMENTIERUNGSKONZEPT

Bei der Beschreibung des Konzepts zur Implementierung der Benutzerschnittstelle ziehen wir uns auf das Niveau einer *abstrakten Implementierung* zurück. Anstelle einer "konkreten" Implementierung mit Datenstrukturen und Algorithmen, die in einer (höheren) Programmiersprache formuliert sind, wird von programmiertechnischen Details abstrahiert: Wir beschränken uns auf die Angabe der *Systemgraph-Tests* und *-Transformationen*, die durch inkrementelle Modifikationen der aktuellen Spezifikationseinheit verursacht werden. Zur formalen Beschreibung einer inkrementellen, syntaxgesteuerten Spezifikations-Erstellung, also von Systemgraph-Transformationen, die die Spezifikation eines "wohlstrukturierten" Softwaresystems in eine ebensolche überführen, dient eine spezielle Graph-Grammatik.

4.3.2.1 GRAPHGRAMMATIK FÜR SYSTEMGRAPH-TRANSFORMATIONEN

Konsistenzerhaltende Systemgraph-Transformationen sind formal durch Produktionen einer *programmierten zweistufigen attributierten Graph-Grammatik* GG_{SG} beschreibbar. Zur Beschreibung von GG_{SG} stützen wir uns auf den Graph-Grammatik-Ansatz von /NAG79/. Danach läßt sich GG_{SG} beschreiben als Tupel

$$GG_{SG} = (\Sigma_{KNO}, \Sigma_{KAN}, KNOTYPEN \cup KNOMARKEN, KANMARKEN, SG_o, P, \text{--- } s \longrightarrow).$$

- Σ_{KNO} und Σ_{KAN} sind zwei endliche, nichtleere Mengen, das Knoten- und Kantenmarkierungs-Alphabet; $KNOTYPEN \cup KNOMARKEN$ ¹⁾ $\subseteq \Sigma_{KNO}$ heißt *terminales* Knotenmarkierungs-Alphabet, $\Sigma_{KNO} - KNOTYPEN \cup KNOMARKEN$ ist das *nichtterminale* Knotenmarkierungs-Alphabet. $KANMARKEN = \Sigma_{KAN}$ ist das *terminale* Kantenmarkierungs-Alphabet (nichtterminale Kantenmarkierungen werden nicht benötigt).
- SG_o ist der Startgraph (SYSTEM).
- P ist eine endliche Menge von *Produktionen* p_i , wobei eine Produktion $p_i = (d_l, d_r, E)$ die Ersetzung eines zu d_l isomorphen und gleichmarkierten Untergraphen einer Graph-Satzform SG_i ²⁾ durch einen anderen Untergraphen d_r unter Beachtung der *Einbettungsüberführung* E beschreibt. Die Einbettungsüberführung³⁾ legt fest, welche Knoten von d_r über welche Kanten mit bestimmten Knoten des Wirtsgraphen SG_i nach der Ersetzung zu verbinden sind.

Beispiele (aus /NAG79/):

$$p = (d_l, d_r, E) \text{ mit}$$

$$d_l = {}^1 \bigcirc_N \quad d_r = {}^2 \bigcirc_a \quad {}^3 \bigcirc_b$$

$$E : l_j = (L_i(1); 3)$$

$$r_k = (2; L_i(1))$$

1) Diese Markierungsmengen wurden in Abschn. 3.2 eingeführt.

2) Eine Graph-Satzform ist ein aus dem Startgraph ableitbarer Graph, der noch nichtterminale Knotenmarkierungen besitzt.

3) In /NAG79/ eine wohlgeformte Zeichenkette.

E definiert, daß sämtliche Knoten des Wirtsgraphen, die (als Quellknoten) durch eine i-Kante mit dem zu ersetzenden Knoten verbunden waren, nach der Ersetzung als Quellknoten über j-Kanten mit dem Knoten 3 von d_r verbunden werden. Außerdem sollen dieselben Wirtsgraph-Knoten Zielknoten von k-Kanten werden, die vom Knoten 2 von d_r ausgehen.

Im folgenden wird in Produktionen die Einbettungsüberführung nur angegeben, falls es sich nicht um die *identische* Einbettungsüberführung handelt. Bei letzterer übernimmt jeder Knoten von d_r die Kantenbezüge des gleichnumerierte Knotens von d_l unverändert.

$\text{--- } s \text{ --->}$ ist die Notation für direkte *sequentielle* Graph-Ableitung. In jedem Ableitungsschritt wird also genau eine Produktion angewendet, d.h. genau ein Untergraph durch einen anderen ersetzt.

Beispiel:

Die Graph-Satzform



wird durch die Produktion p

des obigen Beispiels in den terminalen Graphen überführt.



GG_{SG} ist eine *attributierte* Graph-Grammatik (/GÖT82/): Die Produktionen aus P legen neben der Struktur der neuen Untergraphen auch über *Attribut-Regeln* fest, wie sich in diesem Untergraph die Werte seiner Knotentyp- und Kantenmarkierungs-Attribute berechnen.

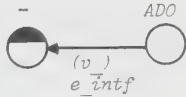
Mit GG_{SG} lassen sich nun über Graph-Satzformen SG_i , die noch nicht-terminale Knotenmarkierungen aufweisen, terminale Graphen ableiten, die "wohlstrukturierte" Softwaresysteme repräsentieren, also Systemgraphen sind.¹⁾

Etliche Produktionen von GG_{SG} werden erst dann anwendbar sein, wenn bestimmte äquivalente Untergraphen der Graphen der linken und rechten Produktionsseite in konsistenter Weise durch einen anderen Untergraph ersetzt worden sind. Die Notwendigkeit, eine *Hyper-Produktion* vor ihrer Anwendung erst mittels einer oder mehrerer *Meta-*

¹⁾ Strenggenommen sind Satzformen, in denen nichtterminale Knoten ausschließlich für optionale Inkremente stehen, bereits Systemgraphen.

Produktionen zu aktualisieren, ergibt sich aus der Zweistufigkeit der verwendeten Graph-Grammatik (/NAG79/). Durch Hyper-Produktionen lassen sich in übersichtlicher Form Sachverhalte beschreiben, wobei von Detail-Aspekten bzw. Feinstrukturen noch abstrahiert wird. Diese Detail-Aspekte oder Feinstrukturen werden dann durch Anwendung der Meta-Produktionen miteingebracht.

Beispiel:

Die bisher verwendete "Abkürzung"  für  entspricht formal einer Meta-Produktion. In dieser Produktion könnte zusätzlich vom konkreten ADO-Bezeichner abstrahiert sein:



Dabei ist die Meta-Knotenmarkierung BEZ¹⁾ noch (gemäß einer weiteren Meta-Produktion) durch einen beliebigen Systemkomponenten-Bezeichner zu ersetzen.

Eine zuvor anzuwendende Meta-Produktion könnte beispielsweise folgende Produktion sein (Darstellung lokaler Systemkomponenten):



Das Ergebnis des Ersetzungsprozesses ist unabhängig von der Reihenfolge, in der die Meta-Regeln angewendet werden.

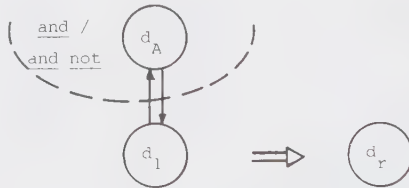
- 1) Meta-Knotenmarkierungen bzw. Meta-Kantenmarkierungen sind durch Unterstreichung, Meta-Knotentypen durch eine doppelte Knoten-Umhüllung gekennzeichnet.
- 2) Gleichartige, jedoch getrennt zu ersetzende Meta-Größen werden indiziert.

Es ergibt sich die Notwendigkeit, bei bestimmten Ableitungen in gewissem Rahmen eine Reihenfolge auf den anzuwendenden Produktionen vorzuschreiben (*programmierte Graph-Grammatik*). Diese Reihenfolge wird in sog. *Kontrollprozeduren* /ENSC82/ vorgegeben. Kontrollprozeduren ähneln Prozeduren einer höheren Programmiersprache und sind aus Graphentests (Bedingungen), Graphproduktionen (Aktionen) oder Aufrufen weiterer Kontrollprozeduren aufgebaut. Der Kontrollfluß wird durch die üblichen Kontrollstrukturen einer höheren Programmiersprache gesteuert.

Um den Kontrollfluß in Kontrollprozeduren überschaubar zu halten, können Produktionen mit *positiven* oder *negativen Anwendbarkeitsbedingungen* (/NAG79/) versehen werden.

Erstere besagen, daß eine Produktion nur dann anwendbar ist, wenn (neben einem zum Untergraphen d_1 der linken Seite isomorphen und gleichmarkierten Untergraphen) auch ein in der Anwendbarkeitsbedingung spezifizierter Untergraph im Wirtsgraphen existiert. Produktionen mit negativen Anwendbarkeitsbedingungen sind dagegen nur anwendbar, wenn außerdem kein Untergraph von der in der Anwendbarkeitsbedingung spezifizierten Form im Wirtsgraphen auftritt.

Graphisch wird der Untergraph d_A einer positiven bzw. negativen Anwendbarkeitsbedingung von dem der linken Seite d_1 durch eine gestrichelte Linie abgetrennt:



Neben Kontrollprozeduren werden auch *Testprozeduren* benötigt. Es handelt sich dabei um boolesche Prozeduren, die jeweils einen umfangreicheren Graphentest durchführen.

Eine programmierte Graph-Grammatik besitzt endlich viele Kontroll- und Testprozeduren.

4.3.2.2 IMPLEMENTATION DES SPEZIFIKATIONS-EDITORS

Der Spezifikations-Editor wickelt den Dialog mit dem Entwickler ab und kontrolliert inkrementelle Modifikationen. Er ist im wesentlichen spezifiziert durch Kontrollprozeduren der Graph-Grammatik, die die inkrementellen Modifikationen durchführen, und einen Kommando-Interpreter zu ihrer Ansteuerung.

Jede durch ein Inkrement-Kommando angestoßene Inkrementoperation wird in einer eigenen Kontrollprozedur spezifiziert. Eine Kontrollprozedur legt neben der operations-spezifischen Kommunikation mit dem Systementwickler auch die durchzuführenden Syntaxprüfungen fest.

Die Kontrollprozedur zu einer Operation auf einem atomaren Inkrement besitzt prinzipiell folgende Grobstruktur:

```
control proc KPROZ [grsymb_nr]
  begin
    Aufbereitung der Eingabezeile;
    Inkrement-Eingabe;
    Konsistenz-Prüfungen;
    Semantische Aktionen;
    /* Positionierung des EDIT-
                                   Zeigers */
    return
  end
```

Der Prozedurkopf besteht nur aus dem Prozedurbezeichner (*KPROZ*). Dieser orientiert sich am entsprechenden Grammatiksymbol (Nummer *grsymb_nr*). Im Prozedurrumpf wird zunächst (bei Eingabe- und Änderungsoperationen) das Bildschirm-Layout gemäß der in Abschnitt 4.3.1.3 beschriebenen Form zur

Abb. 4-11

Durchführung der Operation *aufbereitet* und das neue Inkrement *eingelesen*. Anschließend wird *überprüft*, ob die beabsichtigte Modifikation keine Strukturierungsregel für "wohlstrukturierte" Systeme verletzt. Liegt eine Verletzung vor, wird die Modifikation zurückgewiesen und die Kontrollprozedur verlassen. Wird die Modifikation akzeptiert, werden andere Inkremente, die in irgendeiner Form von der Modifikation betroffen sind, konsistenzerhaltenden Maßnahmen unterzogen. Anschließend erfolgt die Durchführung der Modifikation. Diese *semantischen Aktionen* entsprechen konzeptuell Systemgraph-Transformationen. Vor jedem Rücksprung aus der Kontrollprozedur wird durch ein Unparse-Verfahren die Darstellung der gerade bearbeiteten Spezifikationseinheit auf dem Bildschirm aktualisiert.

Bei Modifikationen atomarer Inkremente besteht die Aktualisierung nur in der Positionierung des EDIT-Zeigers auf das Nachfolge-Inkrement/den Nachfolge-Platzhalter.

Läßt man die Eingabe bzw. die Änderung mehrerer repetitiver atomarer Inkremente als eine Inkrementoperation zu (d.h. die Eingabe bzw. Änderung erfolgt in einer Zeichenkette), wäre der Rumpf der Kontrollprozedur aus Abb. 4-11 in eine repeat-until-Schleife einzubetten.¹⁾

(Obligaten und optionalen) komplexen Inkrementen zugeordnete Kontrollprozeduren unterscheiden sich für Erzeuge- und Löschoptionen etwas.

Im Falle einer Erzeugeoperation wird der Systemgraph um die Knotenstruktur für das neue Inkrement ergänzt. Konsistenzprüfungen sind nicht nötig, da der Spezifikations-Editor an einer bestimmten Position in der aktuellen Spezifikationseinheit die Erzeugung nur solcher Inkremente anbietet, die dort auch konsistent sind. Die Ausgabe der Inkrement-Rahmenstruktur erfolgt durch das mit dem Rücksprung verbundene Unparsing.

Bei einer Löschoption bestehen die semantischen Aktionen aus konsistenzerhaltenden Maßnahmen und der Aktualisierung des Systemgraphen.

Kontrollprozeduren zur Auswahl alternativer komplexer Inkremente²⁾ identifizieren nach der Menue-Ausgabe die vom Entwickler gewählte Alternative und rufen in einer case-Anweisung³⁾ die dieser Alternative zugeordnete Kontrollprozedur auf.

Der *Kommando-Interpreter* des Spezifikations-Editors identifiziert die Inkrement-Kommandos und steuert die jeweilige Kontrollprozedur an. Er ist als erkennender deterministischer Automat realisiert, dessen Zustände den unterschiedlichen Arten komplexer und atomarer Inkremente entsprechen und durch die korrespondierenden nichtterminalen und terminalen Grammatiksymbole notiert sind. In einem bestimmten Zustand werden die für die entsprechende Inkrementart zulässigen Inkrement-Kommandos akzeptiert und die jeweilige Kontroll-

1) Man beachte die Korrespondenz zur {}-Notation in der Grammatik.

2) Alternative atomare Inkremente treten in der Spezifikationssprache nicht auf.

3) Man beachte die Korrespondenz zu den alternativen Regeln in der Grammatik.

prozedur angestoßen. Nach Durchführung der Modifikation wird automatisch in den Folgezustand übergegangen. Dieser ist durch das Grammatiksymbol gegeben, das bei einer pre-order-Traversierung des aktuellen Ableitungsbaums als nächstes angelaufen wird. Führt der Modifikationsversuch zu einer Fehlersituation, verbleibt der Kommando-Interpreter zur Wiederholung der Kommando-Eingabe im aktuellen Zustand.

Ein Zeiger-Kommando bewirkt den direkten Übergang zu dem Folgezustand, der der Art des Zielinkrements entspricht. Bei Eingabe eines falschen Kommandos wird der aktuelle Interpreter-Zustand nicht gewechselt.

Beispiel:

Abb. 4-12 zeigt den Interpreter-Ausschnitt, der die Modifikation einer Operationsspezifikation steuert. Nach Eingabe von **EIN** und der Bereitstellung der Rahmenstruktur durch die Kontrollprozedur *ERZ_OP_SPEZ* wird in den Zustand zur Bearbeitung eines Operations-Bzeichners übergegangen. **LOE** stößt das Löschen der Operation durch *LOE_OP_SPEZ* an. Die Kontrollprozedur positioniert den EDIT-Zeiger im Normalfall vor die nächste Operation, im Fehlerfall beläßt sie ihn vor der nicht gelöschten Operation. In beiden Situationen verbleibt der Interpreter im aktuellen Zustand "op_spez". Zeiger-Kommandos steuern Kontrollprozeduren an, die den EDIT-Zeiger (gemäß ihnen zugeordneten Zeiger-Produktionen) versetzen.

Ein Interpreter-Zustand zur Modifikation eines atomaren Inkrements unterscheidet sich von einem zur Modifikation eines komplexen Inkrements¹⁾ in zwei Punkten. Zum einen wird, falls der EDIT-Zeiger den entsprechenden Platzhalter markiert, sofort die Kontrollprozedur zur Inkrement-Eingabe angesteuert (ansonsten erfolgt die Kommando-Eingabe). Zum anderen ist bei der Kommando-Entschlüsselung das Änderungs-Kommando mitzubberücksichtigen.

In einem Interpreter-Zustand, der die Auswahl unter alternativen Inkrementen kontrolliert, ergibt sich der Folgezustand aus dem von der Kontrollprozedur zurückgemeldeten Auswahl-Ergebnis.

Es besteht die Vermutung, nicht unerhebliche Teile der Kontrollprozeduren und des Kommando-Interpreters des Editors direkt aus

¹⁾ Der im Beispiel beschriebene Zustand war ein solcher.

der formalen Definition der Spezifikationssprache automatisch generieren zu können.

4.3.2.3 ERWEITERUNG DES SYSTEMGRAPHEN

Während der inkrementellen Modifikation der aktuellen Spezifikationseinheit werden Systemgraph-Transformationen nur in dem Untergraph des Systemgraphen durchgeführt, der am AKT_SE-Knoten angehängt ist:

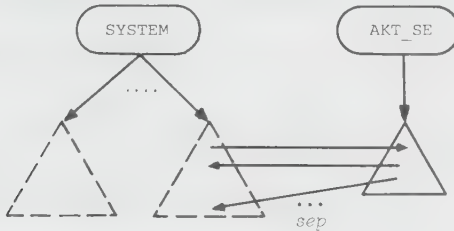


Abb. 4-13

Damit der Systemgraph für Unparse-Zwecke ein direktes Abbild der aktuellen Spezifikationseinheit darstellt, werden die Platzhalter für (komplexe und atomare) Inkremente als *nichtterminale* Knoten abgelegt. Diese Knoten besitzen rechteckige Form und tragen die Bezeichnung des entsprechenden Grammatiksymbols als Inschrift. Die Inschrift der Knoten für *-Platzhalter beginnt mit einem '*', z.B.

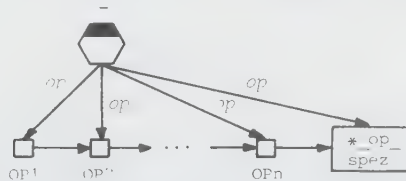
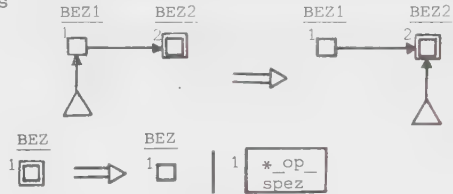


Abb. 4-14: Liste zu exportierender Operationen/Funktionen

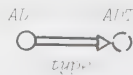
Ein *interner Zeiger*  markiert im Systemgraph den Knoten bzw. die Kante des Inkrements/Platzhalters, auf das am Bildschirm gerade der EDIT-Zeiger verweist.¹⁾ Eine Bewegung des EDIT-Zeigers löst direkt eine korrespondierende Bewegung des internen Zeigers aus. Jedes Zeiger-Kommando führt also zur Anwendung einer speziellen *Zeiger-Produktion*. Mittels



und der Meta-Produktion

wird beispielsweise der Übergang des EDIT-Zeigers zur nächsten Operation bzw. des *-Platzhalters ausgedrückt.

Zeigt der EDIT-Zeiger auf ein Inkrement, das im Systemgraph durch eine Kante repräsentiert wird (beispielsweise eine Typbeziehung), wird das "Deuten" des internen Zeigers auf diese Kante durch Verstärken der Kante dargestellt:



Wird durch den EDIT-Zeiger ein Inkrement markiert, das mehrere Beziehungen gleichzeitig ausdrückt (beispielsweise das Inkrement für den Typ in einer ADO-Deklaration für mehrere abstrakte Datenobjekte (Abb. 2-45)), werden im Systemgraph alle Kanten verstärkt, die die Typbeziehungen repräsentieren.

¹⁾ Die Zustände des Kommando-Interpreters ersparen den internen Zeiger nicht: Der interne Zeiger hält in einer Liste von Inkrementen das aktuelle Listenelement fest; im Kommando-Interpreter ist sämtlichen Listenelementen nur ein Zustand zugeordnet.

4.3.3 BEISPIEL FÜR DIE EINGABE EINER SPEZIFIKATIONSEINHEIT

In diesem Abschnitt wird exemplarisch die inkrementelle Eingabe der äußeren Schnittstelle des GADT's *GT_ASSMEM* demonstriert. Gleichzeitig werden die Zustandsübergänge des Kommando-Interpreters und die Ausführung der Kontrollprozeduren mitbetrachtet.

Der Entwickler setzt zunächst das Kommando für das Erzeugen eines GADT-Rahmens ab. Darauf wird folgende Bildschirmmaske angeboten¹⁾:

▷ uebergeordnete realisierungs-variante

gadt

gadt-bezeichner

(

generische parameter

)

external interface

provides

op

op-bezeichner

(

formale parameter

)

zugriffsrechte/benutzungsrechte

end external interface

realizations

realisierungs-variante

end gadt

Abb. 4-15

¹⁾ Obligate komplexe bzw. atomare Inkremente sind im folgenden stets umrahmt. Auf einem Farb-Bildschirm könnten sie beispielsweise durch entsprechende Farbwahl hervorgehoben werden.

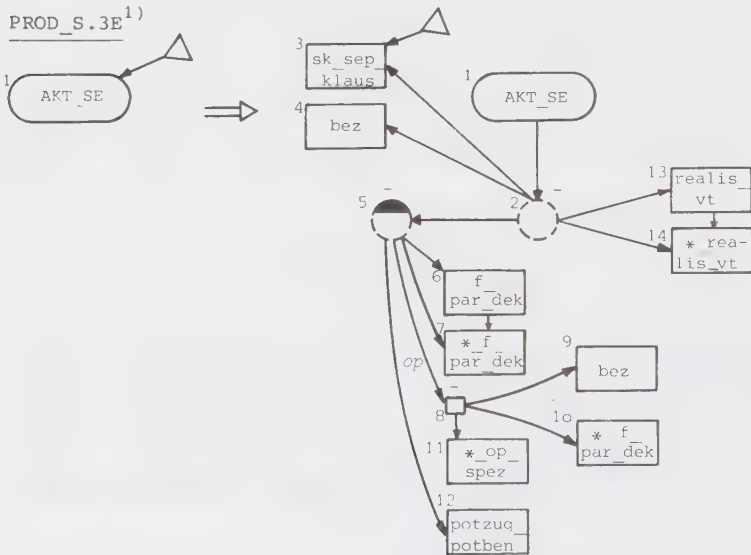
Folgende Kontrollprozedur hat die Überwachung der Bearbeitung übernommen:

```

control_proc ERZ_GADT_DEKL_M_TYDEF [S.3]
  begin Systemgraph-Aktualisierung: PROD_S.3E;
    /* Ausgabe der Rahmenstruktur (Abb. 4-15) */
    return
  end

```

Vor Aufruf dieser Kontrollprozedur ist der mit AKT_SE markierte Knoten aktiviert, beim Rücksprung der Knoten für die noch fehlende separate-Klausel. Die Kontrollprozedur aktualisiert den Systemgraph, indem sie die initiale Knotenstruktur für die aktuelle Spezifikationseinheit aufbaut, und gibt die Rahmenstruktur von Abb. 4-15 aus.



(Nichtterminale) Knoten der Form  repräsentieren entsprechende Platzhalter der Bildschirmmaske. Sie sind über unmarkierte Kanten mit den Knoten für einen GADT bzw. für dessen äußerer Schnittstelle verbunden. Die Knotenmarkierung als Knotenin-schrift charakterisiert die Art der noch einzubringenden Inkremente.

1) Die Numerierung der Produktionen orientiert sich an der Numerierung der Grammatikregeln.

Der interne Zeiger verweist nach Anwendung der Produktion in Korrespondenz zum EDIT-Zeiger auf den Knoten für die (optionale) separate-Klausel.

Da *GT_ASSMEM* als eigenständiges System (und nicht lokal zu einer anderen Systemkomponente) spezifiziert wird, versetzt der Entwickler den EDIT-Zeiger durch einmaliges Betätigen von  gleich vor den Platzhalter 'gadt-bezeichner'. Vom System wird dabei automatisch Platz für die Eingabe des Bezeichners reserviert und der Eingabe-Cursor ■ positioniert:

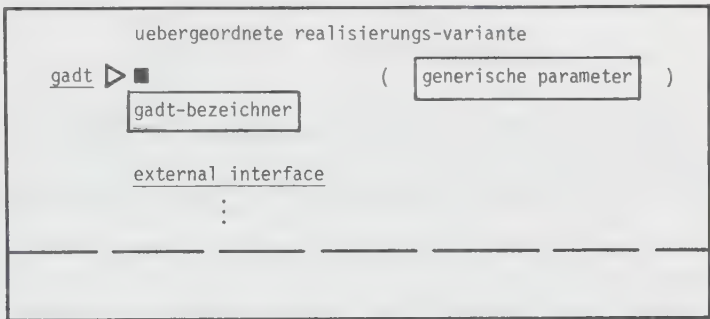
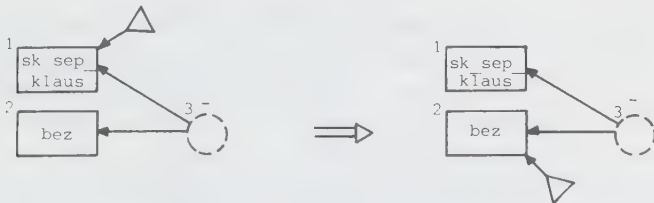


Abb. 4-16

Intern bewirkt das Zeiger-Kommando einen Übergang des Kommando-Interpreters vom Zustand "sk_sep_klaus" in den Zustand "gadt_bez" bzw. das Versetzen des internen Zeigers vom Knoten mit Markierung "sk_sep_klaus" zum (ebenfalls nichtterminalen)¹⁾ Knoten für den GADT-Bezeichner durch die Zeiger-Produktion:



Diese Zeiger-Produktion ist Bestandteil der Kommandoprozedur *POS_AB_SK_SEP_KLAUS*, die zusätzlich den alternativen Fall berücksichtigt, daß der GADT-Bezeichner bereits eingegeben ist.

¹⁾ Zeiger-Produktionen werden im folgenden nicht mehr explizit angegeben.

Nach der Eingabe des Bezeichners *GT_ASSMEM* und dem Eingabe-Abschluß durch **RET** wird der EDIT-Zeiger automatisch zur Eingabe der ersten Deklaration formaler generischer Parameter versetzt:

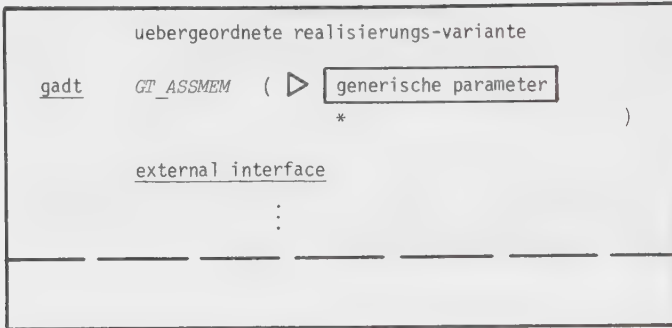


Abb. 4-17

Die Eingabe des GADT-Bezeichners wird von der Kontrollprozedur *EIN_GADT_BEZ* überwacht:

```

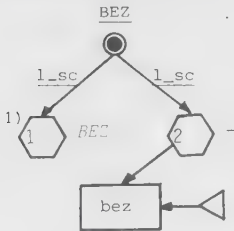
control_proc EIN_GADT_BEZ [S.3.1E]
  begin  Aufbereitung der Eingabezeile (Abb. 4-16);
        Eingabe des GADT-Bezeichners (GADTBZ);
        Test auf Doppeldeklaration:
          if not TEST_DOPP_DEKL_SK (GADTBZ)
            then return (false) fi;
        Systemgraph-Aktualisierung: PROD_S.3.1E.3 (GADTBZ);
        /* Aktualisierung des Bildschirm-Inhalts (Abb. 4-17) */
        return (true)
  end

```

Der eingelesene GADT-Bezeichner *GADTBZ* ist auf eine Doppeldeklaration hin zu überprüfen. Dabei sind zwei Fälle zu unterscheiden:

- a) der Kontext der aktuellen Spezifikationseinheit ist bereits spezifiziert:

TEST_S.3.1E.1 (BEZ)



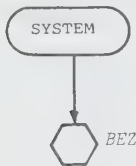
Meta-Produktionen:



Es wird überprüft, ob bei bereits vollzogener Integration (die übergeordnete Spezifikationseinheit existiert schon: erste Alternative der Metaregel) oder bei noch nicht eingebrachter Spezifikationseinheit (zweite Alternative der Metaregel) eine gleichbenannte Bruder-Systemkomponente existiert.

- b) der Kontext der aktuellen Spezifikationseinheit ist noch nicht spezifiziert: TEST_S.3.1E.2 vergleicht den eingegebenen Bezeichner mit den Bezeichnern sämtlicher Systemkomponenten, die noch nicht in andere Systemkomponenten integriert sind.

TEST_S.3.1E.2 (BEZ)



Der Gesamttest wird in folgender Testprozedur durchgeführt:

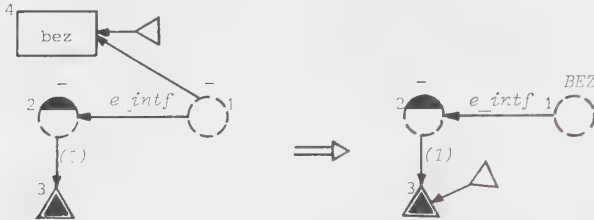
```

test proc  TEST_DOPP_DEKL_SK (BEZ)
  begin if TEST_S.3.1E.1 (BEZ)
    then fehler1; return (false) fi;
    if TEST_S.3.1E.2 (BEZ)
      then fehler2; return (false) fi;
    return (true)
  end
  
```

1) Gleichartige, jedoch getrennt zu ersetzende Meta-Größen werden indiziert.

Liegt keine Doppeldeklaration vor, wird in der Kontrollprozedur EIN_GADT_BEZ die Aktualisierung des Systemgraphen durch folgende Produktion durchgeführt:

PROD_S.3.1E.3 (BEZ)



Meta-Produktionen:



Der interne Zeiger zeigt anschließend auf die (evtl. bereits eingebrachte) erste Deklaration formaler Parameter.

Der Entwickler möchte nun die formalen generischen Parameter von `GT_ASSMEM` spezifizieren. Nach Drücken von **EIN** wird ihm im Informations-Bereich des Bildschirms ein Menue alternativer Inkrementarten angeboten:

```

uebergeordnete realisierungs-variante

gadt  GT_ASSMEM  ( ▷ generische parameter
                        *
                        )

external interface
:
:

```

Alternativen: ■ formale zugriffsrechte
 formale benutzungsrechte
 formale edo-parameter
 formale edt-parameter

Abb. 4-18

Angenommen, der Entwickler positioniert den Eingabe-Cursor ■ durch zweimaliges Drücken von  auf die dritte Alternative. Nach Abschluß durch  wird im Spezifikationstext die Rahmenstruktur für eine Deklaration von EDO-Parametern ausgegeben:

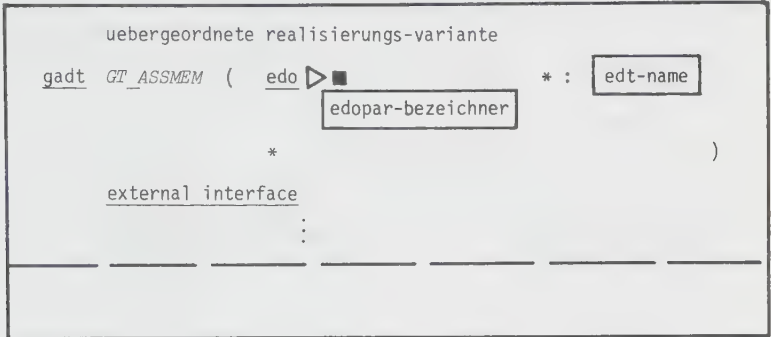


Abb. 4-19

Die Auswahl einer Alternative wird durch folgende, im Interpreter-Zustand "gadt_f_par_dekl" angestoßene Kontrollprozedur abgewickelt:

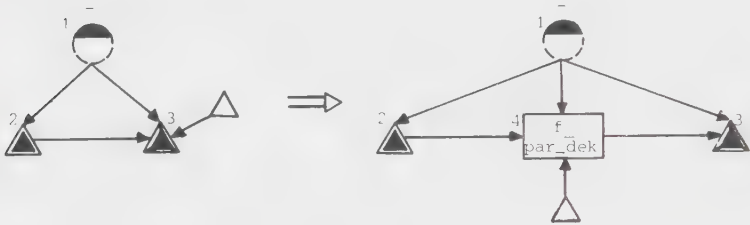
```

control_proc ERZ_GADT_F_PAR_DEKL [S.3.2]
begin if not EDIT-Zeiger vor Platzhalter für Parameter-Deklaration
then Systemgraph-Aktualisierung: PROD_S.3.2E.1
    /* Ausgabe des Platzhalters */
fi;
Menue-Ausgabe (Abb. 4-18);
Auswahl einer Alternative (ALTERN);
case ALTERN
    (f_adopar_dekl): ERZ_GADT_F_ADOPAR_DEKL;
    (f_adtpar_dekl): ERZ_GADT_F_ADTPAR_DEKL;
    (f_edopar_dekl): ERZ_GADT_F_EDOPAR_DEKL;
    (f_edtpar_dekl): ERZ_GADT_F_EDTPAR_DEKL;
end case;
/* Ausgabe der Rahmenstruktur (Abb. 4-19) */
return (ALTERN)
end

```

Falls der EDIT-Zeiger vor dem Inkrement einer Parameter-Deklaration oder dem entsprechenden *-Platzhalter positioniert ist, wird zur Eingabe der neuen Parameter-Deklaration eine neue Zeile mit dem entsprechenden Platzhalter aufbereitet. Im Systemgraph wird in die Kette der Knoten für die bisherigen Parameter-Deklarationen ein neuer (nichtterminaler) Knoten eingekettet:

PROD_S.3.2E.1

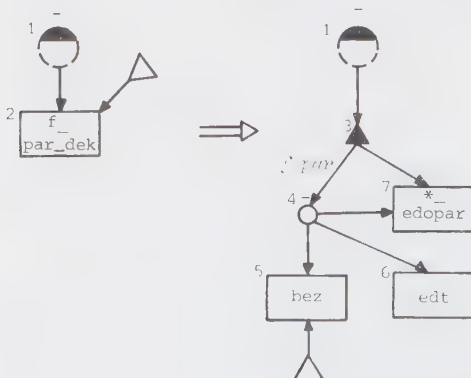


Meta-Produktionen:



Nach der Menue-Ausgabe verzweigt ERZ GADT F PAR DEKL je nach der gewählten Alternative zur Kontrollprozedur für die Erzeugung des entsprechenden alternativen Inkrements. In unserem Fall ist dies ERZ GADT F EDOPAR DEKL. Sie gibt die Rahmenstruktur einer EDO-Parameter-Deklaration gemäß Abb. 4-20 aus und aktualisiert den Systemgraph durch:

PROD_S.3.5E.1



Nach dem Rücksprung aus *ERZ_GADT_F_EDOPAR_DEKL* erfolgt die Rückgabe der Kontrolle von *ERZ_GADT_F_PAR_DEKL* an den Kommando-Interpreter. Dieser geht in den Folgezustand über, der der gewählten Alternative (hier, "edopar") entspricht.

Der Entwickler gibt nun den Bezeichner eines EDO-Parameters ein (Eingabe-Abschluß durch **RET**) :

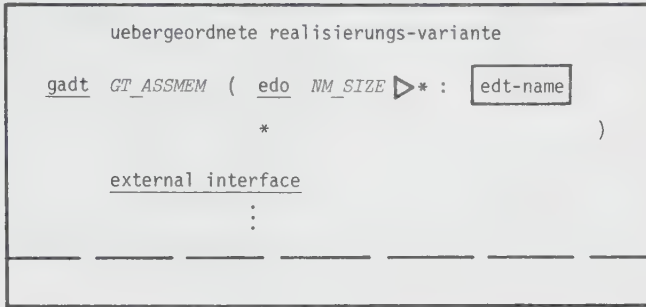
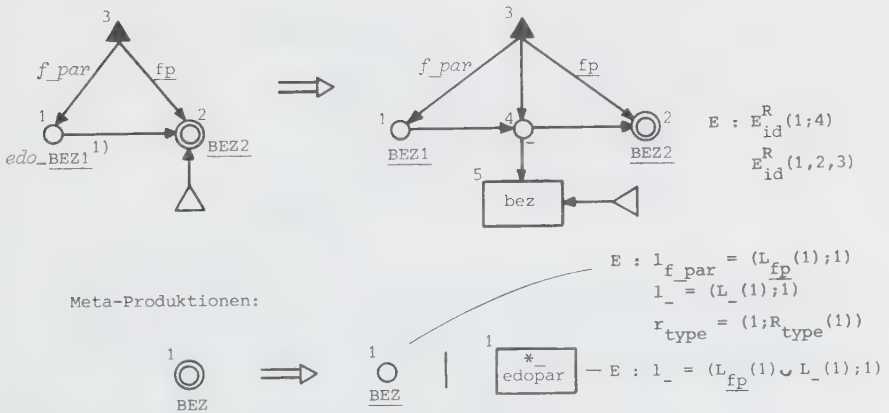


Abb. 4-20

Die Eingabe-Kontrolle übernimmt die Kontrollprozedur *EIN_F_EDOPAR_BEZ*:

Wird der neue Parameter zwischen bereits eingebrachten Parametern eingefügt, wird im Systemgraph die Knotenstruktur für diesen Parameter in die der Parameterliste eingehängt:

PROD_S.3.5.1E.1



¹⁾ Der kontextsensitive Zusatz *edo* verdeutliche, daß hier der Bezeichner eines (formalen) einfachen Datenobjekts gemeint ist.

Zur Einbettungsüberführung von PROD_S.3.5.1E.1:

Allgemein liegt hinsichtlich der Typbeziehung stets die Situation von Abb. 4-19a vor (bei noch nicht spezifiziertem Typ steht anstelle von $\bigcirc_{\text{Typ}}$ der

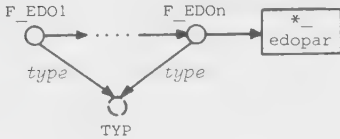


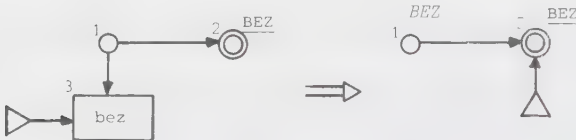
Abb. 4-19 a

Knoten $\boxed{\text{edt}}$). Der Knoten für den neuen EDO-Parameter braucht also nur die entsprechende Beziehung eines Bruderknotens

übernehmen. $E_{id}^R(1;4)$ bedeutet, daß Knoten 4 sämtliche aus Knoten 1 herauslaufende Kanten unverändert übernimmt. Die Einbettung der Knoten 1, 2, 3 wird unverändert übernommen.

Nach der Eingabe der Parameter-Bezeichner wird zunächst durch die Testprozedur TEST_DOPP-DEKL_F_PAR auf eine mögliche Doppeldeklaration getestet. Liegt keine Doppeldeklaration vor, wird der Systemgraph durch PROD_S.3.5.1E.2 aktualisiert:

PROD_S.3.5.1E2 (BEZ)



Möchte der Entwickler nun den Typ des Parameters NM_SIZE spezifizieren, drückt er $\boxed{\rightarrow}$:

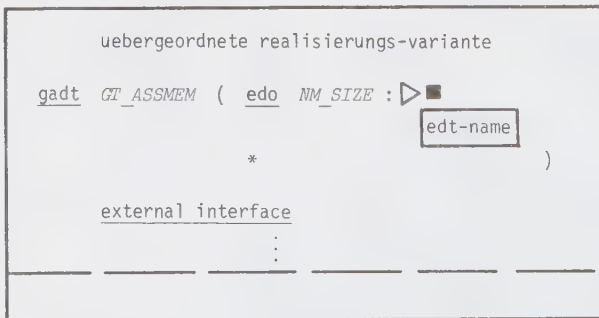


Abb. 4-21

Der Kommando-Interpreter geht dadurch in den Zustand "typ_eines_edopar" über.

Der Typ von NM_SIZE sei INTEGER:

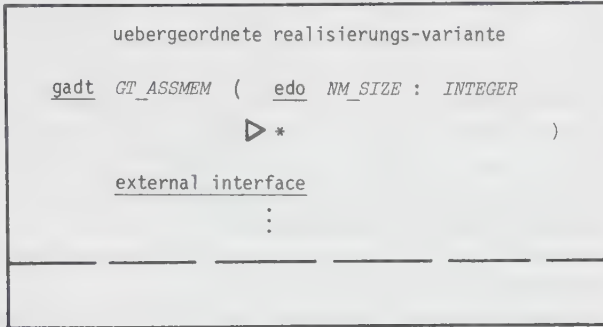


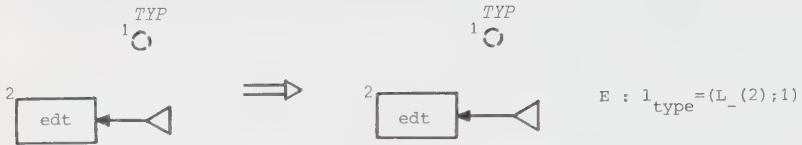
Abb. 4-22

Folgende Kontrollprozedur überwacht die Eingabe des Typs:

```
control proc EIN_F_EDOPAR_TYP [S.3.5.2E]  
  begin Aufbereitung der Eingabezeile (Abb. 4-21);  
    Eingabe des Typ-Namens (TYPNAM);  
    Test auf zulässige Benutzung:  
      if not TEST_TYP_BEN (TYPNAM)  
        then fehler; return (false) fi;  
    Systemgraph-Aktualisierung:  
      Typ-Eintrag für alle EDO-Parameter der  
      Parameterliste : PROD_S.3.5.2E.1 (TYPNAM);  
      Löschen des Platzhalter-Knotens (triviale  
      Produktion);  
      /* Aktualisierung des Bildschirm-Inhalts (Abb. 4-22) */  
      return (true)  
  end
```

Nach der Eingabe des Typs wird durch TEST_TYP_BEN dessen Konsistenz mit evtl. existierenden Benutzungsregeln für einfache Datentypen überprüft (INTEGER sei ein zulässiger Typ). Danach wird für jeden EDO-Parameter die Typbeziehung nachgetragen:

PROD_S.3.5.2E.1 (TYP)



Der Entwickler möchte nun noch die beiden EDO-Parameter DM_SIZE und TM_SIZE, beide ebenfalls vom Typ INTEGER, als zweiten und dritten Parameter nachtragen. Durch Betätigen von und zweimaligem wird der EDIT-Zeiger auf den *-Platzhalter positioniert. Ein bereitet die Eingabestelle gemäß Abb. 4-23 vor.

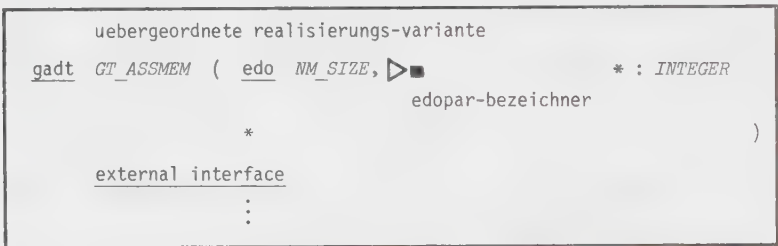


Abb. 4-23

Die Eingabe der zusätzlichen Parameter wird jeweils durch ein abgeschlossen. Nach Eingabe von TM_SIZE liegt folgende Situation vor:

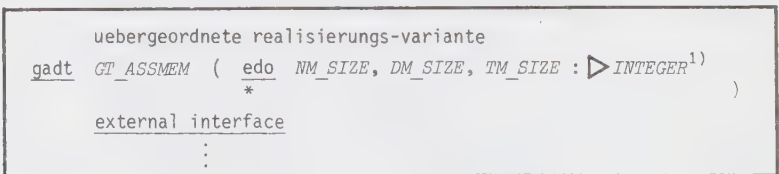
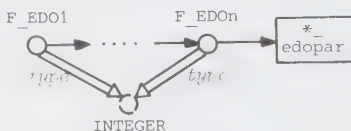


Abb. 4-24

Der Eingabevorgang wird jeweils durch die bereits bekannte Kontrollprozedur EIN_F_EDOPAR_BEZ überwacht.

1) Aktuelle Situation im Systemgraph:



Um nun die GADT-Operationen einzubringen, wird der EDIT-Zeiger durch ,  und dreimaliges  vor die Position des Bezeichners der ersten Zugriffsoperation versetzt.¹⁾ Nach Eingabe des Operationsbezeichners SYSBEGIN ergibt sich folgender Bildschirm Aufbau:

```

uebergeordnete realisierungs-variante

gadt GT_ASSMEM ( edo NM_SIZE, DM_SIZE, TM_SIZE : INTEGER )
  external interface
    provides
      op SYSBEGIN ( ▷ formale parameter
                    *
                    );
    :
    :

```

Abb. 4-25

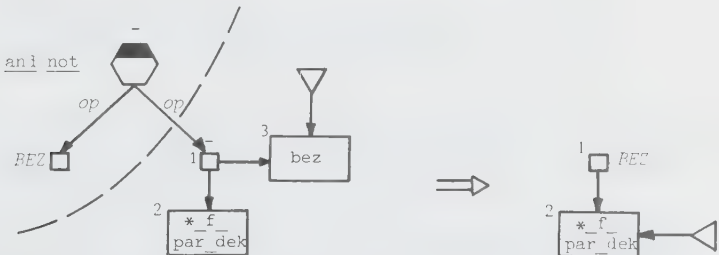
EIN_OP_BEZ steuert die Eingabe eines Operationsbezeichners im Interpreter-Zustand 'op_bez':

```

control proc EIN_OP_BEZ [S.6.1.1E]
  begin
    aufbereitung der Eingabezeile;
    Eingabe des Operationsbezeichners (OPBEZ);
    Test auf Doppel-Deklaration und Systemgraph-Aktualisierung:
      PROD_S.6.1.1E.1 (OPBEZ);
    if Doppel-Deklaration then fehler; return (false) fi
    return (true)
  end

```

wobei PROD_S.6.1.1E.1 (BEZ)



¹⁾ Die Rahmenstruktur der ersten Operation ist bereits angeboten!

Die Eingabe der formalen Parameter einer Zugriffsoperation¹⁾ ähnelt der Eingabe der generischen formalen Parameter eines GADT's. Auf ihre Darstellung wird deshalb verzichtet. Der EDIT-Zeiger deutete anschließend auf den *-Platzhalter für die Spezifikation weiterer Operationen. Durch ein EIN wird die Rahmenstruktur für die zweite Operation erzeugt:

```

:
:
provides
  op SYSBEGIN (          ...          );
  op ► ■ ( formale parameter );
      op-bezeichner
  *
:
:
```

Abb. 4-26

Die korrespondierende Kontrollprozedur:

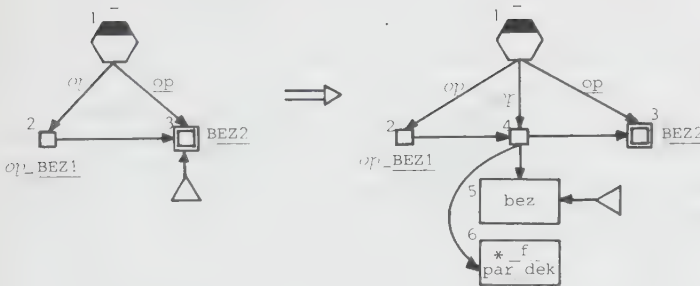
```

control proc ERZ_OP_SPEZ [S.6.1]
  begin Systemgraph-Aktualisierung : PROD_S.6.1E.1;
        /* Ausgabe der Rahmenstruktur (Abb. 4-26) */
  return
end
```

PROD_S.6.1E.1 kettet einen neuen (noch unmarkierten) Knoten in die Liste der Knoten für die bisherigen Operationen ein:

¹⁾ Es sind EDO- bzw. ADO-Parameter zulässig.

PROD_S.6.1E.1



Meta-Produktionen:



$$E : l_{op} = (L_{op}(1) \cup L_{-}(1); 1)$$

$$E : l_{-} = (L_{op}(1) \cup L_{-}(1); 1)$$

Sind die Bezeichner und die formalen Parameter der zweiten Zugriffsoption DECLARE_SVE spezifiziert, wandert der EDIT-Zeiger wieder auf den *-Platzhalter:

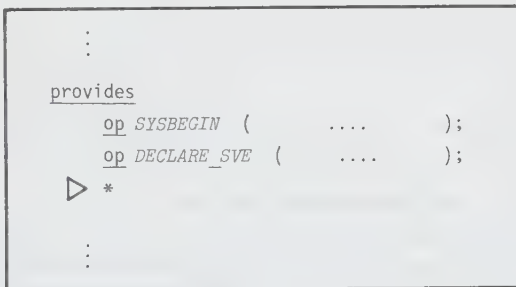


Abb. 4-27

Möchte man zwischen beiden bereits spezifizierten Operationen eine weitere Operation einbringen, würde dort durch  und  die entsprechende Rahmenstruktur bereitgestellt.

Die soweit vorgestellte Beispiel-Eingabe verdeutliche die Benutzer-Schnittstelle hinreichend. In einem späteren Kapitel wird die Implementation weiterer interessanter Inkrementoperationen diskutiert werden.

KAPITEL 5: VON DER SYSTEMSPEZIFIKATION ZUM AUSFÜHRBAREN SYSTEM

5.1 CODIEREN-IM-GROSSEN

Im Rahmen des *Codierens-im-Großen* wird die implementierungssprach-unabhängig formulierte Systemspezifikation¹⁾ in eine äquivalente Form übertragen, die sich ausschließlich der Sprachkonstrukte der für die Systemimplementation gewählten Implementierungssprache bedient. Die Transparenz der resultierenden Spezifikation ist dabei stark davon abhängig, inwieweit die Implementierungssprache das Programmieren-im-Großen konzeptuell unterstützt. Für eine Implementierungssprache, die wie z.B. ADA ein *Modulkonzept* vorsieht, gestaltet sich die Übertragung sicher durchgängiger als für eine Implementierungssprache, die wie z.B. FORTRAN 77 im wesentlichen nur fürs Programmieren-im-Kleinen konzipiert ist.

Die in der Implementierungssprache formulierte Systemspezifikation soll automatisch erstellbar sein und die intendierte Struktur des zu entwickelnden Softwaresystems widerspiegeln.

Bausteine für die physische Modularisierung eines Softwaresystems sind *Programm-Moduln*. Im Zusammenhang mit unserem Software-Arbeitsplatz sind vor allem folgende Eigenschaften eines Programm-Moduls interessant:

Jeder Programm-Modul

- repräsentiert eine Entwurfsentscheidung,
- stellt in seiner *Export-/Import-Schnittstelle* Ressourcen nach außen bereit bzw. führt alle von außen eingeführten Ressourcen auf, die im *Modulrumpf* verwendet werden,
- enthält im Modulrumpf die Implementation der exportierten Ressourcen,
- ist ersetzbar durch einen anderen Programm-Modul mit gleicher Export-/Import-Schnittstelle (und gleicher semantischer Spezifikation), aber anderem Modulrumpf,
- ist getrennt bearbeitbar und getrennt übersetzbar.

¹⁾ Sie kann batchorientiert oder inkrementell erstellt worden sein.

Die äußere Schnittstelle einer Systemkomponente läßt sich somit in die Export-/Import-Schnittstelle, die Realisierungs-Varianten einer Systemkomponente in gegenseitig austauschbare Modulrumpfe übertragen. Die innere Schnittstelle einer Realisierungs-Variante wird

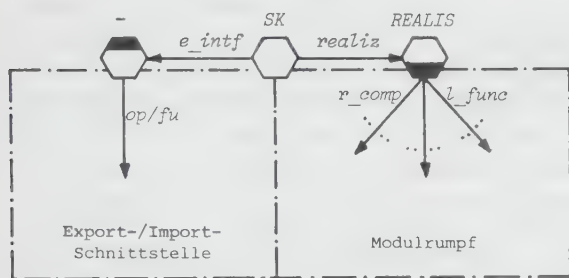


Abb. 5-1

dabei in lokale Daten bzw. Prozeduren des entsprechenden Modulrumpfs umgesetzt. Operationen (bzw. Funktionen) einer Systemkomponente werden als Prozeduren eingebracht. Diese sind in der Export-/Import-

Schnittstelle spezifiziert. Ihre Rumpfe befinden sich im Modulrumpf und werden im Zuge des Programmierens-im-Kleinen ausprogrammiert. Abb. 5-1 zeigt den Zusammenhang zwischen einer Systemkomponente *SK* und dem entsprechenden Programm-Modul in einer Implementierungssprache.

Sind die Export-/Import-Schnittstelle und der Rumpf eines Moduls getrennt übersetzbar¹⁾, resultiert jede Spezifikationseinheit der implementierungssprach-unabhängigen Spezifikation in einer eigenen Übersetzungseinheit der Implementierungssprache. Generell sollte jede Spezifikationseinheit, selbst wenn sie nicht getrennt eingebracht wurde, in eine getrennt zu übersetzende Übersetzungseinheit übertragen werden. Dies ermöglicht effizientere Maßnahmen zur Wiederherstellung der Konsistenz des implementierten Systems, falls nach der Implementierung Modifikationen an der Systemspezifikation vorzunehmen sind.

Die Übertragung der Spezifikation einer Systemkomponente kann sofort erfolgen, sobald sie sich als korrekt erweist.

Bietet die Implementierungssprache ein Modulkonzept, läßt sich das skizzierte Schema eines Programm-Moduls (*Modulschema*) mehr oder weniger direkt durch die konkrete Struktur eines Programm-Moduls

¹⁾ Davon wird im folgenden ausgegangen.

der Implementierungssprache realisieren. Die Realisiert-, Zugriffsrecht- und Benutzungsrecht-Beziehung der implementierungssprachunabhängigen Spezifikation sind in entsprechende Modulbeziehungen des Modulkonzepts umzusetzen bzw. durch deren disziplinierte Verwendung nachzubilden. Kennt die Implementierungssprache kein Modulkonzept, sind Modulschema und Modulbeziehungen mit den gegebenen Sprachkonstrukten zu "simulieren".

Die Informationen der implementierungssprachlichen Spezifikation werden vom Software-Arbeitsplatz natürlich nicht eigens abgespeichert, sondern bei Bedarf jeweils aus dem Systemgraph gewonnen.

IPSEN sieht in seiner ersten Ausbaustufe MODULA-2 als Ziel-Implementierungssprache vor. Generell könnten daneben auch andere Zielsprachen angeboten werden. In den folgenden Abschnitten wird die Umsetzung der Spezifikation anhand der beiden Sprachen ADA und FORTRAN 77, die gewissermaßen Extremfälle in ihrer Eignung zur Implementierung modularer Systeme darstellen, diskutiert.

5.1.1 ÜBERTRAGUNG NACH ADA

ADA bietet fürs Programmieren-im-Großen komfortable Konstrukte (/DoD80/, /NAG82/).

ADA sieht ein Modulkonzept, das sog. Paketkonzept, vor. Jeder Programm-Modul, ein Paket, ist textuell in die Paketschnittstelle (Export-/Import-Schnittstelle) und den Paketrumpf (Modulrumpf) getrennt.

Eine Paketschnittstelle zerfällt in einen sichtbaren Teil und einen privaten Teil. Im sichtbaren Teil werden alle Ressourcen spezifiziert, die exportiert, also für andere Pakete sichtbar sein sollen. Der private Teil enthält Deklarationen, die zwar aufgrund des Prinzips der Informationshiding von importierenden Paketen abzuschotten sind, deren Kenntnis jedoch der ADA-Compiler benötigt. Ein Paketrumpf enthält die Implementation der exportierten Ressourcen.

Paketschnittstelle und Paketrumpf sind getrennt übersetzbar. Informationen zu getrennt eingebrachten Übersetzungseinheiten hält die ADA-Programmierumgebung in der sog. Systemstruktur-Datei (library file), Quell- und Binärfassungen der Übersetzungseinheiten in der sog. Programmsystem-Bibliothek (program library). Mit Hilfe des Konzepts des generischen Pakets lassen sich (zur Übersetzungszeit) beliebig viele Instanzen von Paketen erzeugen, die sich nur in strukturellen Details unterscheiden. Pakete unterliegen der Gültigkeitsbereichs-Regelung des Blockstrukturkonzepts von ADA: Ein Paket ist nur in dem Block gültig,

der das Paket einführt. Das Paket-Innere selbst ist wiederum blockstrukturiert.

ADA kennt zwei Arten von Beziehungen zwischen Paketen: die Enthaltenseins-Beziehung und die Benutzt-Beziehung. Ein Paket P1 enthält ein Paket P2, falls P2 lokal im Rumpf von P1 deklariert wird. Der Rumpf von P2 heißt Untereinheit zu P1, falls P1 P2 enthält und der Rumpf von P2 getrennt übersetzt wird. Ein Paket P1 kann ein Paket P2 benutzen¹⁾, falls P2 gewissermaßen auf äußerstem Blockstrukturniveau deklariert ist und allgemein verwendbare Ressourcen anbietet. P2 heißt Bibliothekseinheit. Die in einem Paket benutzbaren Bibliothekseinheiten werden in einer Kontextspezifikation (sog. with-Klausel), die der Paketschnittstelle vorangestellt ist, bekanntgemacht.

Wie lassen sich nun die verschiedenen Arten von Systemkomponenten in ADA spezifizieren? In Abb. 5-2 ist eine ADO-Deklaration mittels implizitem ADT dargestellt (vgl. Abb. 2-2 bzw. Abb. 4-2).

Das ADO *ASSMEM* wird als gleichnamige Bibliothekseinheit spezifiziert, wobei Schnittstelle und Rumpf dieses Pakets getrennt übersetzt werden. Während sich die Paketschnittstelle direkt aus der äußeren Schnittstelle von *ASSMEM* in der implementierungssprach-unabhängigen Spezifikation ableitet (1), ergibt sich in der Spezifikation des Paketrumpfs eine leichte Änderung. Lokale Systemkomponenten (hier *NM* u.a.) werden als lokale Pakete (Paket-Hierarchie) spezifiziert. Die Schnittstellen dieser Pakete werden an den Deklarationsstellen aufgeführt (2). Nur die zugehörigen Paketrümpfe sind als eigene Untereinheiten textuell ausgelagert (3)²⁾. In Abb. 5-2 sind auch bereits die Rahmen der in der Implementierungsphase noch auszuprogrammierenden Prozeduren berücksichtigt (4).

Die *ASSMEM*-Operationen sind in anderen Paketen mit Hilfe der sog. Punktnotation referenzierbar, z.B. *ASSMEM.SYSBEGIN(...)*.

Der ADT *T_ASSMEM*, der in der implementierungssprach-unabhängigen Spezifikation mittels Typdefinition deklariert wurde, wird in ADA ebenfalls als Bibliothekseinheit spezifiziert (Abb. 5-3). Es wird ein Paket eingeführt (*T_ASSMEM_P*), das in seiner Schnittstelle den Typ *T_ASSMEM* exportiert. Da die Repräsentierung des Typs nach außen hin verborgen bleiben soll, wird *T_ASSMEM* als privater Typ

1) Diese Benutzt-Beziehung unterscheidet sich von der Benutzungsrecht-Beziehung des in Kapitel 2 eingeführten Strukturmodells.

2) Das separate in unserer Spezifikationssprache und in ADA besitzen gleiche Bedeutung.

```
-- ado

package ASSMEM is                                     (1)
    procedure SYSBEGIN (...);
    procedure DECLARE_SVE (...);
    :
    procedure SYSEND (...);
end ASSMEM;

-- ado-realization REALIS1
package body ASSMEM is

    -- internal interface

    -- representation
    package NM is                                       (2)
        :
        :
    end NM
    package body NM is separate; (3)
    :
    :

    -- local data types

    -- local functions
    procedure RELEASE_ENT (...) is
    end RELEASE_ENT;

    -- end internal interface

    -- implementation
    procedure SYSBEGIN (...) is (4)
    end SYSBEGIN;
    :
    procedure SYSEND (...) is
    end SYSEND;

    -- end implementation

end ASSMEM;
```

Abb. 5-2: ADA-Paket für eine ADO-Deklaration mittels implizitem ADT

spezifiziert (3). Exportiert wird somit neben den *T_ASSMEM*-Operationen nur der Typbezeichner. Im privaten Teil der Paketschnittstelle (4) ist nun bereits die Repräsentierung von *T_ASSMEM* anzugeben (5), die bei der Spezifikation des ADO's *ASSMEM* (Abb. 5-2) noch im Pakettrumpf steckte. Die Repräsentierung ist als Verbundtyp-Definition zu formulieren (6), in der natürlich (im Gegensatz zur Repräsentierung von *ASSMEM*) keine Pakete auftreten dürfen. Sämtliche Komponenten der Repräsentierung, die mittels implizitem ADT eingeführt werden sollten, sind deshalb über einen expliziten ADT zu deklarieren (7), wobei die den Verbundkomponenten übertragenen Operationen in Kommentarfom angeführt werden (8). Die zur Deklaration der Verbundkomponenten verwendeten abstrakten Datentypen (*T_NM* u.a.) sind (wie *T_ASSMEM*) in eigenen Paketen zu spezifizieren. Daß diese abstrakte Datentypen an der Stelle ihrer Verwendung in der Typrepräsentierung sichtbar sein müssen, zwingt zu einer einschneidenden Umstrukturierung: Die Pakete zu diesen abstrakten Datentypen können nicht lokal im Rumpf von *T_ASSMEM_P* deklariert werden, sondern sind - obwohl ohne allgemein verwendbaren Charakter - als eigene Bibliothekseinheiten einzubringen, die dann von *T_ASSMEM_P* benutzt werden. Diese Benutzt-Beziehung¹⁾ wird in entsprechenden with-Klauseln hergestellt (1). Die Realisiert-Beziehung der implementierungssprach-unabhängigen Spezifikation läßt sich also im vorliegenden Fall leider nicht in die Enthaltenseins-Beziehung (also keine Paket-Hierarchie möglich), sondern "nur" in eine Benutzt-Beziehung zwischen ADA-Paketen umsetzen, die allerdings kontrolliert verwendet wird. Use-Klauseln (2) erlauben es, die von den benutzten Paketen exportierten abstrakten Datentypen direkt, d.h. ohne über die Punktnotation, zu referenzieren.

Der Aufbau des getrennt übersetzten Pakettrumpfs von *T_ASSMEM_P* ist - abgesehen von der Auslagerung der Repräsentierung - mit dem Pakettrumpf von *ASSMEM_P* identisch.

Der von *T_ASSMEM_P* exportierte ADT *T_ASSMEM* kann nun von einem beliebigen Anwendungssystem *APPL_SYS* aktuell benutzt werden, z.B. in

1) Diese hat mit der Benutzungsrecht-Beziehung der implementierungssprach-unabhängigen Spezifikation nichts gemein.

```
-- adt

with T_NM_P; (1)      use T_NM_P; (2)
:
with T_CBUF_P;      use T_CBUF_P;

package T_ASSMEM_P is
  type T_ASSMEM is limited private;      (3)
  procedure SYSBEGIN (...);
  procedure DECLARE_SVE (...);
  :
  procedure SYSEND (...);

private      (4)
  type T_ASSMEM is      (5)
    -- representation
    record      (6)
      NM : T_NM;      (7)
      -- provides OP1, OP2, ... (8)
      :
      CBUF : T_CBUF;
      -- provides ...

    end record;

end T_ASSMEM_P;
```

```
-- ado-realization REALIS1
package body T_ASSMEM_P is

  -- internal interface

  -- local data types
  -- T_NM, ..., T_CBUF
  :
end T_ASSMEM_P;
```

Abb. 5-3: ADA-Paket für eine ADT-Deklaration mittels Typdefinition

einer ADO-Deklaration mittels explizitem ADT (Abb. 5-4). Die in der implementierungssprach-unabhängigen Spezifikation vorgenommene Abmagerung des *T_ASSMEM*-Operationensatzes wird in Kommentarförm wiedergegeben:

```
with T_ASSMEM_P; use T_ASSMEM_P;
package APPL_SYS_P is
  :
  ASSMEM : T_ASSMEM
    -- provides SYSBEGIN, DECLARE_MVE, ...
  :
  :
```

Abb. 5-4: ADO-Deklaration mittels explizitem ADT
in ADA

Zugriffe auf das ADO *ASSMEM* sind dann in der Form "SYSBEGIN (*ASSMEM*, ...)" möglich, d.h. sämtliche *T_ASSMEM*-Operationen haben einen Parameter des Typs *T_ASSMEM* vorzusehen.

GADT-Deklarationen mittels Typdefinition werden als generische Pakete eingebracht:

```
-- gadt

with
  :

generic
  NM_SIZE, DM_SIZE, TM_SIZE : INTEGER;

package GT_ASSMEM_P is
  type GT_ASSMEM is limited private;
  :
  :
end GT_ASSMEM_P;
```

Abb. 5-5: GADT-Deklaration mittels Typdefinition
in ADA

Der Aufbau des generischen Pakets ist - abgesehen von den vorangestellten generischen Parameter - mit dem für einen ADT (Abb. 5-3) identisch. Die generischen Parameter treten in der Definition des Typs *GT_ASSMEM* angewandt auf.

T_ASSMEM kann nun als Typkonkretisierung von *GT_ASSMEM* eingeführt werden:

```
package T_ASSMEM_P is new GT_ASSMEM_P (NM_SIZE => 1, ...); (1)
use T_ASSMEM_P; (2)
subtype T_ASSMEM is GT_ASSMEM; (3)
-- provides ...
```

Abb. 5-6: ADT-Deklaration mittels explizitem GADT in ADA

Zunächst wird (zur Übersetzungszeit) eine neue Paket-Instanz *T_ASSMEM_P* geschaffen (1). Diese Paket-Instanz stellt den Typ *GT_ASSMEM* bereit, dessen Repräsentierung nun endgültig fixiert ist. Er ist also mit dem angestrebten ADT identisch. Um dies auch in seinem Bezeichner zu dokumentieren, benennen wir ihn in *T_ASSMEM* um, wozu wir uns (etwas zweckentfremdend) des Untertyp-Konzepts bedienen (3). Durch eine use-Klausel bzgl. des Pakets *T_ASSMEM_P* war *GT_ASSMEM* direkt sichtbar gemacht worden (2).

Wollte man das ADO *ASSMEM* direkt mittels *GT_ASSMEM* deklarieren, wäre Zeile (3) in Abb. 5-6 gegen "*ASSMEM : GT_ASSMEM*" auszutauschen. Dagegen kann eine GADT-Deklaration mittels explizitem GADT, also eine partielle Absättigung der generischen Parameter, in ADA nicht ausgedrückt werden. Hier müßte eine Fehlermeldung abgesetzt werden, die den Systementwickler zu einer entsprechenden Modifikation der Systemspezifikation auffordert.

Abb. 5-5 zeigte die Spezifikation formaler generischer EDO-Parameter. Die Spezifikation formaler generischer ADO-Parameter sieht genauso aus. Formale generische EDT- und ADT-Parameter werden als private Typen eingeführt, wobei bei ADT-Parametern auch die ihnen übertragenen Operationen zu spezifizieren sind:

```
-- gadt

generic
    type F_EDT is private;
    type F_ADT is private;
    with procedure OP1 (...);
    :
package GADT_P is
    :
    :
```

Abb. 5-7

Die Umsetzung von Funktionsklassen in entsprechende ADA-Pakete ist unkompliziert.

Hinsichtlich der Umsetzung der Zugriffsrecht-Beziehung nach ADA gilt: Alle abstrakten Datenobjekte, die im Strukturmodell im Zugriffsbereich einer Systemkomponente *SK* liegen, sind in der in ADA formulierten Spezifikation im Paket zu *SK* aufgrund der ADA-Blockstruktur auch sichtbar. In diesem Paket sind somit auch generell alle Zugriffe möglich, die in der implementierungssprach-unabhängigen Spezifikation *SK* in Form von Zugriffsrechten eingeräumt wurden.

Zugriffe zwischen Paketen, die wie T_NM_P und T_CBUF_P in Abb. 5-3 als Bibliothekseinheiten einzubringen waren, sind über entsprechende with-/use-Klauseln zu ermöglichen. Die einem Paket aus der implementierungssprach-unabhängigen Spezifikation übertragenen ZR-Attribute werden zu Dokumentationszwecken als Kommentar in diesen with-/use-Klauseln oder (wie in Abb. 5-8) in der Paketschnittstelle aufgeführt. Die Konsistenz zwischen aktuellen Zugriffen und entsprechenden Zugriffsrechten ist natürlich erst in der Implementierungsphase überprüfbar.

Das über Zugriffsrechte Gesagte gilt für Benutzungsrechte analog. Realisierungs-Varianten bedeuten, daß gegenseitig austauschbare Paketrümpfe einzubringen sind.¹⁾

¹⁾ Im Fall von Abb. 5-3 sind sogar austauschbare Paketschnittstellen das Resultat.

```
-- ado

package ADO1 is
  procedure OP1 (...);
  :
  procedure OPn (...);

  -- access/use rights
  -- ado ADO2 by OP3, ...
  :

end ADO1;
```

Abb. 5-8

Die in ADA formulierte Systemspezifikation ist automatisch erstellbar, da der Systemgraph alle dazu nötigen Informationen enthält (u.a. auch die Informationen der Systemstruktur-Datei der ADA-Programmierungsumgebung). Die erstellten Skelette der Paketrümpfe sind Vorgabe für das Programmieren-im-Kleinen. Die-Erstellung der Paketschnittstellen ist i.a. jedoch erst nötig, wenn das fertig entwickelte System in Quelltext-Form auf den Zielrechner übertragen werden soll: Die bei der Übersetzung eines Paketrumpfes benötigten Informationen der zugehörigen Paketschnittstelle liegen ja bereits im Systemgraph vor.

5.1.2 ÜBERTRAGUNG NACH FORTRAN 77

Da FORTRAN 77 kein Modulkonzept anbietet, sind Programm-Moduln und deren Beziehungen durch disziplinierten Gebrauch der FORTRAN-Konstrukte nachzubilden. Die disziplinierte Verwendung wird vom Software-Arbeitsplatz unterstützt und überwacht.

Die von einer Systemkomponente bereitgestellten Zugriffsoperationen bzw. Funktionen werden als SUBROUTINE-Unterprogramme realisiert. Eingebettet in einen Kommentarraahmen, der die äußere und innere Schnittstelle der Systemkomponente dokumentiert, bilden sie einen FORTRAN-"Programm-Modul".

```
C  ** ado1) ASSMEM

C      ** external interface

C          * provides
C              op SYSBEG(...)
C              :
C              :
C              op SYSEND(...)

C          * access/use rights

C      end external interface

C      ** ado-realization REALIS1

C          * internal interface

C              * representation
C                  ado NM : separate
C              :
C              :

C          * local data types

C          * local functions
C              SUBROUTINE RELENT(...)
C              END

C      end internal interface

C          ** implementation
C              SUBROUTINE SYSBEG(...)
C              END
C              :
C              SUBROUTINE SYSEND(...)
C              END

C      end implementation

C      end ado-realization REALIS1

C  end ado ASSMEM
```

Abb. 5-9: FORTRAN-"Modul" für eine ADO-Deklaration
mittels implizitem ADT

¹⁾ Die nicht FORTRAN-konforme Kleinschreibung wird hier aus Lesbarkeitsgründen verwendet.

Die Repräsentierung eines (mittels implizitem ADT deklarierten) ADO's wird als benannter COMMON-Block eingebracht, den alle zu diesem ADO gehörenden Unterprogramme beinhalten. Als COMMON-Block-Elemente sind jedoch nur Repräsentierungs-Komponenten zu berücksichtigen, die in der implementierungssprach-unabhängigen Spezifikation über einen expliziten (G)ADT oder EDT deklariert wurden. Die Repräsentierung von Komponenten, die mittels implizitem ADT deklariert wurden, geht nämlich bereits in den COMMON-Block zu ihren eigenen Unterprogrammen ein.

Beispielsweise werden alle Komponenten der *ASSMEM*-Repräsentierung mittels implizitem ADT deklariert (Abb. 2-2). Die SUBROUTINE-Unterprogramme zu *ASSMEM* enthalten deshalb keinen COMMON-Block für die *ASSMEM*-Repräsentierung (Abb. 5-9). Die Repräsentierung der *ASSMEM*-Komponente *CBUF* (Kommunikationspuffer) könnte aus einem INTEGER-Feld und zwei Zeigern auf den Beginn und das Ende der gepufferten Information, also aus einfachen Datenobjekten, bestehen. Die Unterprogramme für die *CBUF*-Zugriffsoperationen operieren somit auf dem COMMON-Block für die *CBUF*-Repräsentierung, z.B.

```
SUBROUTINE INCBUF(...)
      COMMON /CBUFC/ CBUF(100), INFBEG, INFEND
      INTEGER CBUF(100), INFBEG, INFEND
      :
      :
END
```

Da FORTRAN 77 keine Typdeklarationen kennt, sind Objekte eines expliziten (G)ADT's in der FORTRAN-Spezifikation als INTEGER-Felder zu deklarieren, wobei entsprechende Speicherplatz- und Adreßberechnungen bereits bei der Übertragung nach FORTRAN vorzunehmen sind. Die Größe des INTEGER-Feldes zu einem Objekt ergibt sich aus dem Speicherbedarf für die Repräsentierung eines Objekts des (G)ADT's. Dieser ist iterativ aus dem Speicherbedarf der Komponenten der Repräsentierung berechenbar.

Steht beispielsweise der Typ eines Assoziativspeichers *T ASSMEM* zur Verfügung und wird in einem Anwendungssystem *APPL SYS* ein Assoziativspeicher-Objekt benötigt, wird dieses im COMMON-Block der Repräsentierung des Anwendungssystems berücksichtigt:

```
SUBROUTINE APPLOP (...)  
    COMMON / APPLC / ..., ASSMEM (1000), .....  
    :  
    :  
    INTEGER ASSMEM (1000)  
    :  
    :  
END
```

Der Speicherbedarf eines Assoziativspeicher-Objekts ergibt sich aus der *T_ASSMEM*-Repräsentierung und betrage hier 1000 Wörter.

Die FORTRAN-Spezifikation für einen ADT sieht für die Typrepräsentierung natürlich keinen COMMON-Block vor: Speicherplatz ist nur für Objekte dieses ADT's zu reservieren. Die SUBROUTINE-Unterprogramme für die Zugriffsoperationen des ADT's bekommen die COMMON-Objekte, auf die sie aktuell anzuwenden sind, als Parameter übermittelt.

```
C ** adt T_ASSMEM  
:  
:  
:  
C ** implementation  
    SUBROUTINE SYSBEG(FASSMM, ...)  
        INTEGER FASSMM (1000)  
        :  
        :  
        END  
        :  
C    end implementation  
  
C    end realization REALIS1  
  
C end adt T_ASSMEM
```

Abb. 5-10: FORTRAN-"Modul" für eine ADT-Deklaration

Abb. 5-10 zeigt die Deklaration von *T_ASSMEM*. Die Spezifikation der äußeren und inneren Schnittstelle ähnelt der von *ASSMEM* (vgl. Abb. 5-9). Die Unterprogramme werden über den formalen Parameter *FASSMM* aktuell mit einem als *INTEGER*-Feld realisierten *T_ASSMEM*-Objekt versorgt. Angenommen, die Repräsentierung eines *T_ASSMEM*-Objekts mit einem Gesamt-Speicherbedarf von 1000 Wörtern setzt sich aus

200 Wörtern für die Repräsentierung des Namensspeichers *NM* vom Typ *T_{NM}*, aus 300 Wörtern für die des Datenspeichers *DM* vom Typ *T_{DM}* usw. zusammen. Dann können den SUBROUTINE-Unterprogrammen zu den Typen *T_{NM}*, *T_{DM}* usw. die Repräsentierungs-Komponenten von *T_{ASSEM}* über entsprechende Basisadressen weitergereicht werden. Beispielsweise würde eine *T_{DM}*-Operation DMOP mit der Spezifikation

```
SUBROUTINE DMOP (FDM, ...)
  INTEGER FDM (300)
  :
END
```

den Feldbereich für den Datenspeicher eines Assoziativspeicher-Objekts von einer *T_{ASSEM}*-Operation durchgereicht bekommen durch

```
CALL DMOP (FASSMM (201), ...).
```

Funktionen einer Funktionsklasse werden ebenfalls als SUBROUTINE-Unterprogramme spezifiziert, Zugriffsrechte zwischen Systemkomponenten werden über Parameter ihrer Operationen/Funktionen realisiert.

DM (vom ADT *T_{DM}*) und *CBUF* (vom ADT *T_{CBUF}*) sind Komponenten der *T_{ASSEM}*-Repräsentierung. *T_{DM}* besitze ein Zugriffsrecht bzgl. *CBUF*. Da *CBUF* "nur" Komponente einer Typrepräsentierung ist, sind die *T_{DM}*-Operationen (außer mit einer *T_{DM}*-Inkarnation) auch mit (der Basisadresse) der korrelierenden *T_{CBUF}*-Inkarnation zu versorgen:

```
SUBROUTINE DMOP (FDM, FCBUF, ...)
  INTEGER FDM (300), FCBUF (100)
  :
END
```

Der entsprechende Aufruf:

```
CALL DMOP (FASSM (201), FASSMM (901), ...)
```

Eine automatische Übertragung der implementierungssprach-unabhängigen Systemspezifikation nach FORTRAN 77 ist generell möglich, obwohl evtl. benötigte FORTRAN-Konzepte fehlen. Beispielsweise ist bei Verwendung dynamischer Datenstrukturen der Haldenspeicher im FORTRAN-Programm zu simulieren. Bei Verwendung einfacher Datentypen wären bei der Übertragung Speicherplatz- und Adreßberechnungen (wie bei abstrakten Datentypen) durchzuführen.

5.2 IMPLEMENTIEREN EINES PROGRAMM-MODULS

Sobald die Spezifikation einer Systemkomponente *SK* in die eines entsprechenden Programm-Moduls der gewählten Implementierungssprache umgesetzt ist, kann der Modulrumpf¹⁾ ausprogrammiert werden (Programmieren-im-Kleinen). Dem Systementwickler werden dabei neben den Spezifikationen der zu implementierenden Prozeduren auch alle Objekte der implementierungssprachlichen Systemspezifikation vorgegeben, auf die in der Prozeduren-Implementation zurückgegriffen werden darf: Datenobjekte/-typen und Prozeduren, die Systemkomponenten der implementierungssprach-unabhängigen Spezifikation realisieren und bzgl. denen *SK* ein Zugriffs- bzw. Benutzungsrecht besitzt. Ressourcen der inneren Schnittstelle von *SK* werden dabei direkt im auszuprogrammierenden Modulrumpf, Ressourcen im Zusammenhang mit expliziten Zugriffs- bzw. Benutzungsrechten in Informationsform vorgegeben.

Analog der Erstellung der Systemspezifikation sind auch für die Ausprogrammierung eines Modulrumpfes zwei Eingabe-/Änderungsmodi vorgesehen: ein batchorientierter und ein inkrementeller/syntaxgesteuerter Implementierungsmodus.

Bei der *batchorientierten* Vorgehensweise wird der Programmtext der Prozeduren-Implementation mit Hilfe des Texteditors erstellt bzw. geändert. Anschließend wird der Modulrumpf von einem Compiler für die Implementierungssprache *übersetzt*. Der Compiler, der auf die Informationen des Systemgraphen zurückgreift, überprüft die syntaktische Korrektheit der Prozeduren-Implementation^{2),3)} und baut den *Modulgraph* /ENSC82/ als ihre interne Repräsentierung auf. Obwohl der Modulgraph im Übersetzungsprozeß nur als Zwischensprache dient, wird er auch nach der Übersetzung aufbewahrt: Der Programmtext der implementierten Prozeduren wird - analog dem Quelltext der Systemspezifikation - nicht direkt aufbewahrt, sondern bei Bedarf aus dem Modulgraph rekonstruiert. Abb. 5-11 zeigt das Zusammenspiel zwischen Systemgraph und einem Modulgraph:

- 1) Im folgenden beziehen wir uns anstelle konkreter Modulstrukturen in Implementierungssprachen stets auf das Schema eines Moduls (vgl. Abschnitt 5.1).
- 2) Insbesondere die Konsistenz hinsichtlich Zugriffs- und Benutzungsrechten.
- 3) Die Ressourcen der inneren Schnittstelle braucht der Compiler nicht überprüfen, da sie Vorgaben sind. Sie sind jedoch bei der Codeerzeugung mitzubetrachten.

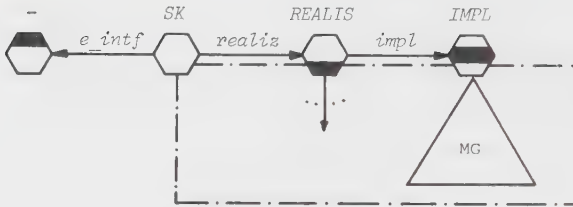


Abb. 5-11

Die *impl*-Kante drückt aus, daß *IMPL* eine Implementation der Operationen (bzw. Funktionen) der Systemkomponente *SK* zur Realisierungs-Variante *REALIS* ist. In ihr können also die Ressourcen verwendet werden, die von der inneren Schnittstelle von *REALIS* bereitgestellt werden. Diesen "Überbrückungs"-Aspekt drücke die Lage des schwarzen Balkens im Knoten mit Markierung *IMPL* aus. Dieser Knoten bildet die Wurzel des Modulgraphen zur Implementation *IMPL*. Die —·—·—·-Umrahmung verdeutliche, welche Ressourcen der übersetzte Modulrumpf enthält.

Entsprechend den Vorgaben, die dem Entwickler zur Eingabe der Prozeduren-Implementation gemacht werden, existiert vor der Eingabe bereits ein Modulgraph-Skelett: Der Modulgraph ist mit den Untergraph-Strukturen für die Spezifikation der *SK*-Operationen (bzw. -Funktionen) initialisiert:

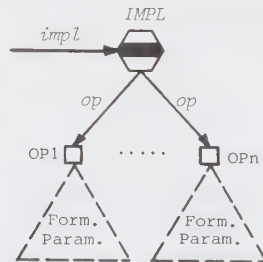


Abb. 5-12

Der initiale Modulgraph wird während des Syntaxanalyse-Prozesses um die Untergraphen für die Rümpfe der Operationen ergänzt. Darin auftretende aktuelle Zugriffe werden gemäß Abb. 5-13 vermerkt:

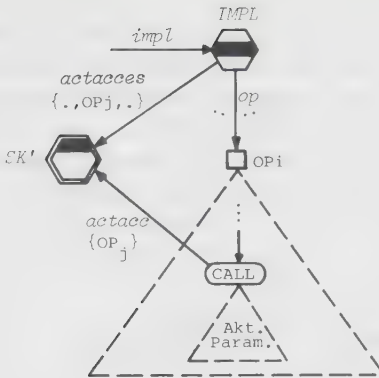


Abb. 5-13

Ein aktueller Zugriff, syntaktisch ein Prozeduraufruf, werde im Modulgraph an der Stelle seines Auftretens durch einen CALL-Knoten beschrieben. Von diesem Knoten wird eine *actacc*-Kante (actual access) zum Knoten der äußeren Schnittstelle des Zugriffsobjekts *SK'* im Systemgraph gezogen. Die verwendete Zugriffsoperation wird als Attributwert der Kantenmarkierung notiert. Die in den Rümpfen der Operationen insgesamt auftretenden aktuellen Zugriffe bzgl. *SK'* werden in einer *actaces*-Kante (actual accesses) zusammengefaßt, die vom Knoten der Implementation zum Knoten für die äußere Schnittstelle von *SK'* führt. Diese "technische" Kante ermöglicht effizientere konsistenzhaltende Maßnahmen, falls Zugriffsoperationen modifiziert werden.

Aktuelle Benutzungen von (generischen) abstrakten Datentypen zur Deklaration von abstrakten Datenobjekten, die in den *SK*-Operationen/-Funktionen lokal benötigt werden, werden entsprechend den aktuellen Benutzungen in der äußeren oder inneren Schnittstelle von *SK* notiert (vgl. Abschn. 2.2.4.4). Es können also *type*- bzw. *g_type*-Kanten aus dem Modulgraph zum Knoten des aktuell benutzten (G)ADT's im Systemgraph führen. Sämtliche derartigen Bezüge sind wieder in einer *actuses*-Kante aufsummiert.

Aktuelle Aufrufe von (P)FK-Funktionen lassen sich ähnlich den aktuellen Zugriffen beschreiben.

Die Konsistenz zwischen aktuellen Zugriffen/Benutzungen und den entsprechenden Rechten ließe sich in speziellen Konsistenzbedingungen ausdrücken. Auf die Angabe dieser Bedingungen wird jedoch verzichtet.

Erweist sich die zu übersetzende Prozeduren-Implementation als syntaktisch korrekt, bestimmt der Compiler bei datenorientierten Systemkomponenten den Speicherbedarf für jede Repräsentierungs-Komponente eines einfachen Datentyps und vermerkt ihn am Knoten der Komponente als Attributwert.¹⁾ Außerdem erzeugt der Compiler aus dem Modulgraph für jede implementierte Prozedur Bindecode, der über eine *code*-Kante an die Modulgraph-Darstellung der Prozedur angehängt wird:

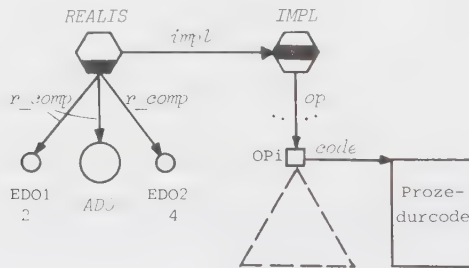


Abb. 5-14

Legt man im Laufzeit-Datenbereich eines übersetzten Modulrumpfs MR1 die Repräsentierungs-Komponenten von einfachem Datentyp vor denen eines abstrakten Datentyps ab, sind diese einfachen Datenobjekte im Prozedurcode bereits relativ zur Basisadresse der Repräsentierung adressierbar. Dies gilt nicht für Komponenten eines abstrakten Datentyps, da dessen Repräsentierung in einem anderen Modulrumpf MR2 eingeführt werden kann, der evtl. noch nicht übersetzt ist. Eine Modifikation dieser Repräsentierung wird deshalb auch keine Neuübersetzung von MR1 erfordern.

Modifikationen an den ausprogrammierten Operationenrumpfen von MR1 erfordern eine Neuübersetzung des Modulrumpfs: Modulgraph und Bindecode sind neu zu erstellen.

Neben dem batchorientierten Implementierungsmodus sieht das Konzept unseres Software-Arbeitsplatzes auch einen *inkrementellen* und *syntaxgesteuerten* Entwicklungsmodus vor. Der Entwickler program-

¹⁾ Der Speicherbedarf für Komponenten eines abstrakten Datentyps ist in dessen Repräsentierung festgehalten.

miert die Prozedurrümpfe inkrementweise aus, wobei die Inkremente syntaktischen Einheiten der gewählten Implementierungssprache entsprechen. Ähnlich den Inkrementen der Spezifikationssprache lassen sich komplexe und atomare Inkremente unterscheiden. Eingegebene Inkremente werden von einem *inkrementellen Compiler* (/SCH75b/, /NAG80/) auf ihre Korrektheit hinsichtlich kontextfreier und kontextsensitiver Syntax überprüft und im Modulgraph repräsentiert. Für alle ausführbaren Inkremente, die als syntaktisch korrekt erkannt wurden, erzeugt der inkrementelle Compiler Codestücke:

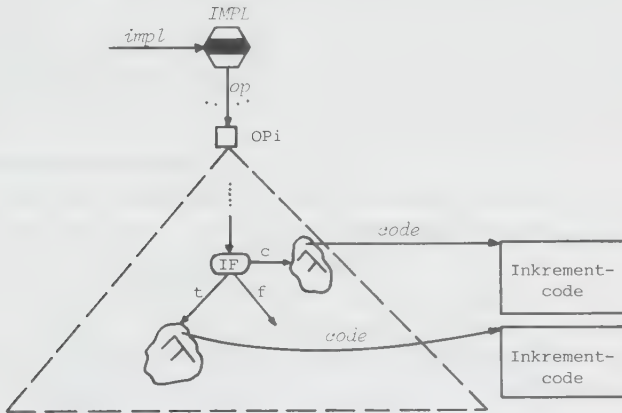


Abb. 5-15

Die implementierten Prozeduren können nun ausgeführt werden, indem der Modulgraph *interpretiert* wird und Inkremente, für die Codestücke existieren, ausgeführt werden (*interpretative Prozedurausführung*). Die Implementation einer Prozedur braucht somit nur insofern vollständig zu sein, als sämtliche Inkremente auf dem bei der Ausführung durchlaufenen Programmpfad eingebracht sind. Wird ein noch unvollständiges Inkrement angelaufen, wird der Systemlauf zur Vervollständigung des Inkrements unterbrochen. Wird ein Inkrement geändert, bedeutet dies die Neuübersetzung des Inkrements und weiterer Inkremente einer begrenzten Nachbarschaft: Mit Hilfe des Modulgraphen ist exakt bestimmbar, welche anderen Inkremente von der Modifikation betroffen sind. Nur für diese und

das modifizierte Inkrement sind Untergraphen und Codestücke zu ersetzen. Analog sind beim Löschen eines Inkrements nur Inkremente einer begrenzten Umgebung konsistenznerhaltenden Maßnahmen zu unterziehen.

/SCH75b/, /BRE77/, /NAG80/ und /ENSC82/ zeigten auf, daß programmierte Graph-Grammatiken ein exzellentes Hilfsmittel zur Beschreibung eines inkrementellen Compilers sind.

Der inkrementelle Compiler ist mit einem *syntaxgesteuerten Programm-Editor* kombiniert: Der Entwickler spezifiziert über Kommandos die Art des einzugebenden Inkrements. Für komplexe Inkremente kann somit sofort die Rahmenstruktur vorgegeben werden. Atomare Inkremente können sofort gegen die spezifizierte syntaktische Kategorie abgeprüft werden. Zusätzlich wird der Entwickler durch Menuetechnik über alle syntaktisch möglichen Eingaben informiert.

Wir lassen zu jeder Realisierungs-Variante *REALIS* einer Systemkomponente *SK* mehrere Varianten an Prozeduren-Implementationen (*Implementations-Varianten*) zu:

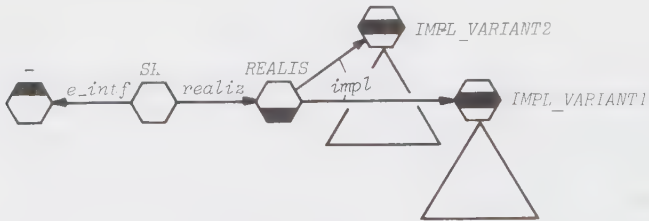


Abb. 5-16

Die Varianten können beispielsweise von der Verwendung unterschiedlicher Implementierungssprachen (ADA-Variante, FORTRAN-Variante), einem unterschiedlichen Teststadium (Dummy-Variante, ungetestete bzw. voll ausgetestete Variante), algorithmischen Aspekten (laufzeitoptimale oder speicherplatzoptimale Variante) usw. herrühren.

Die Charakteristika einer Implementations-Variante werden durch benutzerdefinierte *Implementations-Attribute* beschrieben (vgl. Realisierungs-Varianten in Abschn. 2.2.5). Jede Implementations-Variante ergibt zusammen mit den Datenobjekten/-typen der inneren Schnittstelle von *REALIS* eine Modulrumpf-Variante.

5.3 SYSTEMAUSFÜHRUNG

5.3.1 AUSFÜHRUNGSMODI

Der Software-Arbeitsplatz IPSEN sieht drei alternative Modi der Systemausführung vor, die jeweils einen der Testabschnitte Modultest, Test der Modulinteraktionen und Systemtest im besonderen unterstützen.

Zum einen kann das System *konventionell* ausgeführt werden: Der Code der übersetzten Modulrumpfe wird gebunden, die gebundene Binärfassung des Systems geladen und gestartet. Der Binder ist dabei durch Angabe von Attributen steuerbar, die für die resultierende ausführbare *System-Variante* gelten sollen (*System-Attribute*): Er berücksichtigt nur Bindecodes von Modulrumpfen (oder Teilsystemen), die Realisierungs- bzw. Implementations-Varianten mit den vorgegebenen Attributen implementieren.¹⁾

Für jeden selektierten Modulrumpf bestimmt der Binder die endgültige Größe des Speicherbereichs, der im Laufzeit-Datenspeicher für modullokale Objekte der Repräsentierungen benötigt wird²⁾, und innerhalb dieses Speicherbereichs die Relativadressen lokaler abstrakter Datenobjekte. Dazu wertet er, die Modulhierarchie bottom-up traversierend, die Speicherbedarfs-Angaben aus, die im Systemgraph zu den Repräsentierungen in den selektierten Modulrumpfen gemacht sind. Sodann wird der Ladecode erstellt, indem prozedurrelative Adressen in systemrelative Adressen gewandelt und Bezüge zwischen Prozeduren abgesättigt werden. Der Systemgraph-Knoten für die Systemwurzel verweist auf die Ladecodes sämtlicher bereits erstellter System-Varianten:

¹⁾ Zur Selektion der Varianten könnte man sich sogar eine eigene Sprache für Attribut-Ausdrücke vorstellen.

²⁾ Bei der Bindecode-Erzeugung brauchte der Speicherbedarf von lokalen abstrakten Datenobjekten noch nicht bekannt sein. Im folgenden wird aus Gründen der Vereinfachung generell davon ausgegangen, daß dieser Speicherbedarf bereits zur Bindezeit vollständig berechnet werden kann.

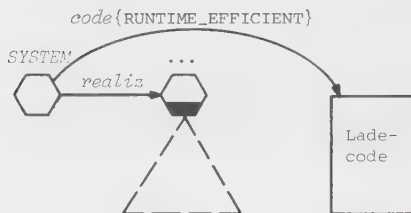


Abb. 5-17

Vor dem Systemstart wird der Lade-code einer System-Variante geladen. Der vom Binder statisch verwaltete Teil des LZ-Datenspeichers (modullokale Objekte) wird beim Systemstart angelegt. Er bleibt während der gesamten Systemausführung erhalten. Im sich anschließenden dynamisch verwalteten Teil des LZ-Datenspeichers werden zur Laufzeit prozedurlokale Datenobjekte (Laufzeit-Keller) und Haldenobjekte angelegt. Der LZ-Datenspeicher ist in Abb. 5-18 skizziert (vgl. die Repräsentierung in Abb. 5-14):

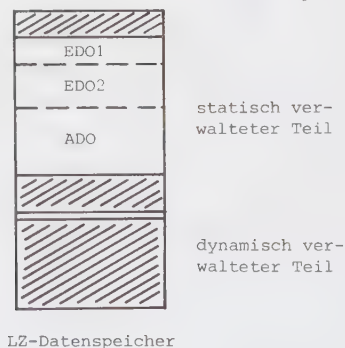


Abb. 5-18

Nach jeder Modifikation eines Modulrumpfs und seiner anschließenden Neuübersetzung ist ein neuerliches, relativ zeitaufwendiges Binden des Gesamtsystems nötig. Während die konventionelle Systemausführung deshalb wohl erst im Zuge des Systemtests für die Erstellung einer System-Variante bedeutsam ist, die gegen die Anfor-

derungsdefinition zu testen ist, benötigt man für den Testabschnitt, in dem (nach dem Test der Einzelmoduln) der Test der Modulinteraktionen ansteht (Testen-im-Großen), einen flexibleren Ausführungsmodus. In diesem Teststadium sind beispielsweise einige Teilsysteme oder Prozeduren noch nicht implementiert oder gesperrt, da sie noch zu modifizieren sind. Trotzdem möchte man bereits andere Teilsysteme testen. Die sog. *interpretative Systemausführung* bietet die gewünschte Flexibilität. Dieser Ausführungsmodus beruht auf dem Konzept, die Systemstruktur anhand des Systemgraphen zur Laufzeit interpretativ abzuarbeiten.¹⁾ Ein *Systemstruktur-Interpretierer* ermöglicht es zum einen, das Binden der von den Moduln exportierten Prozeduren bis zur Laufzeit hinauszuschieben (dynamisches Binden). Erst beim erstmaligen Aufruf einer Prozedur ist ihr Code zu laden, was die aktuelle Belegung des Laufzeit-Programmspeichers auf ein Mindestmaß reduziert. Die Basisadresse des geladenen Prozedurcodes wird für spätere Bindevorgänge im Modulgraph über eine *c_addr*-Kante (*c*ode *b*ase *a*ddress) vermerkt:

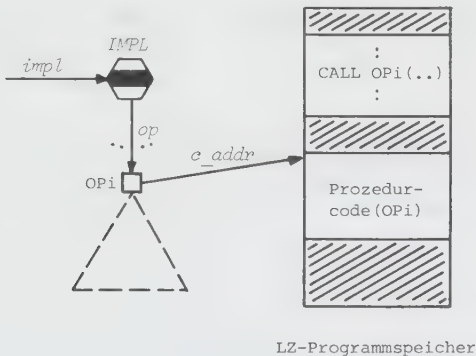


Abb. 5-19

Beim Aufruf einer Prozedur *OPI* übergibt die rufende Prozedur die Kontrolle an den Systemstruktur-Interpretierer, der anhand des Modulgraphen die Basisadresse des geladenen Codes von *OPI* bestimmt. Dabei kann kurzfristig zwischen den Prozedurcodes aus verschiede-

¹⁾ Ein interpretativer Ausführungsmodus ist in konventionellen Programmiersystemen nicht vorgesehen.

nen Implementations-Varianten gewählt werden. Sobald der Befehlszähler mit einer Code-Basisadresse geladen ist, geht die Kontrolle an die gerufene Prozedur über.

Zum anderen erlaubt es der interpretative Ausführungsmodus, den von einem Modulrumpf benötigten Speicherbereich im LZ-Datenspeicher erst dynamisch zu bestimmen und anzulegen.¹⁾ Zur dynamischen Speicherbedarfs-Berechnung wird die Modulhierarchie (analog der statischen Berechnung durch den Binder im konventionellen Ausführungsmodus) interpretativ ausgewertet. Anschließend wird der Speicherbereich für die modullokalen Objekte der Repräsentierungen im LZ-Datenspeicher angelegt. Die resultierenden Objekt-Adressen werden im Systemgraph durch d_addr -Kanten (data object address) vermerkt. Im Prozedurcode werden Objekte der Repräsentierungen indirekt (über Systemgraph-Verweise) adressiert. Abb. 5-20 skizziert diese dynamische Adressierungstechnik.²⁾

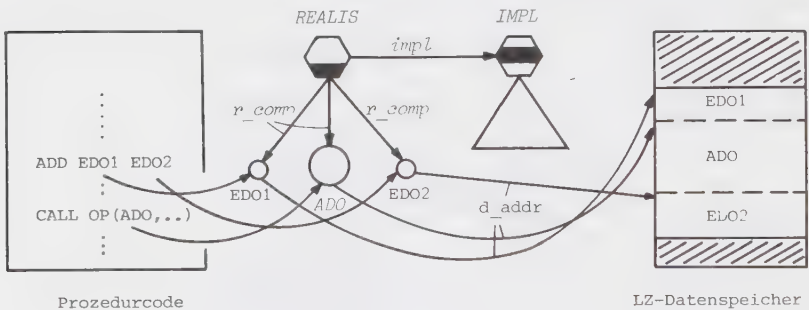


Abb. 5-20

LZ-Datenspeicher- und LZ-Programmspeicher sind bei einer interpretativen Ausführung stets minimal belegt. Für den Nachteil einer erhöhten Laufzeit aufgrund des interpretativen Vorgehens tauscht man komfortable Testmöglichkeiten ein. Das System kann trotz unvollständiger Teilsysteme ausgeführt werden, solange der Kontroll-

1) Dies geschieht beim erstmaligen Aufruf einer Prozedur dieses Moduls.

2) Es wird also nun auch der im konventionellen Ausführungsmodus statisch verwaltete Datenspeicher-Teil dynamisch verwaltet.

fluß diese nicht berührt. Wird trotzdem eine noch nicht implementierte Prozedur angelaufen, wird der Systemlauf unterbrochen, die Prozedur implementiert und anschließend der Systemlauf am Unterbrechungspunkt wieder aufgesetzt. Analoges gilt für Fehlerkorrekturen, die aufgrund von Laufzeit-Fehlern an Prozedur-Implementationen vorzunehmen sind. Zudem kann bei Prozeduraufrufen dynamisch eine gegenüber der Voreinstellung alternative Realisierungs- oder Implementierungs-Variante gewählt werden. In gewissem Rahmen sind auch zur Laufzeit noch Modifikationen an der Systemspezifikation möglich, sofern die Konsistenz des lauffähigen Systems wieder hergestellt werden kann.

Für den Test der Modulinteraktionen erwies sich eine interpretative Systemausführung als nützlich, wobei die Prozeduren der Moduln konventionell, d.h. direkt ausgeführt werden. Im Zusammenhang mit dem Test von Einzelmoduln ist jedoch die Kombination der interpretativen Systemausführung mit einer *interpretativen Prozedurausführung* wünschenswert. Die Interpretation eines Modulgraphen als Intern-darstellung eines inkrementell übersetzten Modulrumpfs eröffnet zusätzliche komfortable Möglichkeiten für den Modultest /NAG80/.

5.3.2 AUSFÜHRUNGSÜBERWACHUNG

Für eine verbesserte Fehlererkennung bzw. gezielte Fehlersuche stellt der vorgestellte Software-Arbeitsplatz fürs Programmieren-im-Großen und fürs Programmieren-im-Kleinen Testhilfen in Form einer komfortablen Ausführungsüberwachung bereit. Da der Systemgraph und die Modulgraphen auch zur Laufzeit des Systems existieren, ist eine spezifikationssprach- bzw. implementierungssprach-bezogene Überwachung möglich. Systemgraph und Modulgraphen sind jedoch dazu statisch und auch dynamisch um testspezifische Zusatzinformationen zu erweitern.¹⁾ Speziell fürs Programmieren-im-Großen lassen sich mindestens folgende Testfunktionen vorsehen:

¹⁾ Auf diese wird hier nicht weiter eingegangen.

- a) Setzen von Unterbrechungspunkten: Es können statisch (insbesondere im batchorientierten Ausführungsmodus) spezifikationsbezogene Unterbrechungspunkte definiert werden, an denen die Kontrolle an den Entwickler zurückgegeben wird. Solche Unterbrechungspunkte könnten z.B. durch die faktischen Zugriffe auf bestimmte abstrakte Datenobjekte gegeben sein. Dabei könnte zusätzlich bzgl. einzelner Zugriffsoperationen selektiert, nur jeder n-te Zugriff berücksichtigt oder zu berücksichtigende Zugriffe an bestimmte Zugriffs-Subjekte (beispielsweise eines ausgewählten Teilsystems) gekoppelt werden. Faktische Aufrufe von Funktionen lassen sich ähnlich überwachen. Bei Erreichen eines Unterbrechungspunkts wird der entsprechende Zugriff/Aufruf im Programm-Text z.B. durch Blinken angezeigt.
- b) Schnappschüsse und Ablaufverfolgung: Nach einer Unterbrechung des Systemlaufs können Schnappschüsse von der aktuellen Belegung bestimmter ADO-Repräsentierungen (oder Teilen davon) angefordert werden. Die Aufbereitung ihrer Ausgabe orientiert sich so nah wie möglich an der Systemspezifikation. Eine Ablaufverfolgung (Trace) kann die letzten faktischen Zugriffe/Aufrufe protokollieren. Ähnlich dem Setzen von Unterbrechungspunkten ist eine selektive Auswahl zu berücksichtigender Zugriffe/Aufrufe vorgesehen. Gleichzeitig mit einem faktischen Zugriff könnten die dadurch verursachten Veränderungen an der aktuellen Belegung der Repräsentierung des Zugriffs-Objekts angezeigt werden. Daneben kann auch die Protokollierung zukünftiger Zugriffe/Aufrufe eingeschaltet werden.
- c) Wiederaufsetzen der Ausführung: Nach gewissen Modifikationen an der Systemspezifikation/-implementation sollte die Systemausführung auch an früher durchlaufene Wiederaufsetzpunkte "zurückgespult" werden können. Die Wiederaufsetzpunkte sind statisch, aber auch dynamisch (z.B. letzter verändernder Zugriff auf bestimmtes ADO) definierbar.

Die geschilderten Testhilfen sollten dynamisch durch ein Kommando unterdrückt und zu einem späteren Zeitpunkt geschlossen oder teilweise reaktiviert werden können.

KAPITEL 6: NACHTRÄGLICHE INKREMENTELLE MODIFIKATION DER SYSTEM-SPEZIFIKATION

6.1 DURCHFÜHRUNG EINER MODIFIKATION

Noch in der Entwurfsphase können sich aufgrund von Umstrukturierungen, Ergänzungen u.ä. Modifikationen an der bereits eingebrachten implementierungssprach-unabhängigen Spezifikation als notwendig oder nützlich erweisen. Aber auch Probleme oder Fehlersituationen, die während der Implementierung oder während des Testens auftreten, können Modifikationen an der Spezifikation nahelegen bzw. sogar erforderlich machen. Ebenso kann die Pflege eines bereits integrierten und getesteten Systems (beispielsweise aus Optimierungsgründen) noch zu Modifikationen führen.

Jede Modifikation an der Spezifikation führt i.a. zu Inkonsistenzen innerhalb einer gewissen Umgebung in der Systemspezifikation und (sofern bereits vorhanden) auch in der Systemimplementation¹⁾ (*Wirkbereich* einer Modifikation). Die Wirkbereiche von Modifikationen an der äußeren Schnittstelle einer Systemkomponente divergieren in zwei Richtungen: Während Modifikationen an Zugriffs- und Benutzungsrechten sich höchstens in dem von der Systemkomponente aufgespannten Teilsystem (mit den entsprechenden Moduln) auswirken, sind bei Modifikationen an der Spezifikation exportierter Operationen/Funktionen Systemkomponenten (bzw. die entsprechenden Moduln) einer begrenzten Nachbarschaft außerhalb betroffen. Modifikationen innerhalb einer Realisierungs-Variante einer Systemkomponente können Inkonsistenzen bei lokalen Systemkomponenten bzw. in der Implementation der Prozeduren nach sich ziehen. Die äußere Schnittstelle der Systemkomponente - und damit auch das Restsystem - sind von diesen Modifikationen nicht berührt. Vereinzelte, "kleinere" Modifikationen wird der Entwickler vermutlich inkrementell, umfangreichere Modifikationen wohl eher batch-orientiert²⁾ durchführen. Da erstere Modifikationen i.a. häufiger

1) Die Möglichkeit, daß sich eine Modifikation auch auf die Implementation von Programm-Moduln auswirkt, ist der Grund dafür, daß diese Diskussion nicht in ein früheres Kapitel vorgezogen wurde.

2) Es werden in einer Spezifikationseinheit größere Inkremente als eine Zeichenkette eingegeben.

auftreten, wollen wir die Diskussion nachträglicher Modifikationen auf inkrementelle Modifikationen beschränken.

Im Gegensatz zu konventionellen Programmiersystemen, bei denen der Entwickler den Wirkbereich einer Modifikation selbst feststellen muß, bzw. im Gegensatz zu Software-Entwicklungsumgebungen, bei denen der Wirkbereich einer Modifikation bei getrennter Übersetzung nur auf der Ebene der potentiellen Zugriffe zwischen Übersetzungseinheiten bestimmt wird, kann der vorgestellte Software-Arbeitsplatz den Wirkbereich exakt, nämlich auf der Ebene der Inkremente bestimmen und entsprechende konsistenzerhaltende Maßnahmen einleiten. Die durch eine inkrementelle Modifikation verursachten Inkonsistenzen hängen stark von der Art des modifizierten Inkrements ab. In einigen Fällen sind sie (je nach dem Ausführungsmodus) durch eine automatisch angestoßene Neuübersetzung betroffener Module und neuerlichem Binden oder aber durch Neuübersetzung betroffener Programm-Inkremente behebbar. Meistens jedoch sind sie nur von den Entwicklern selbst zu beseitigen, die für die inkonsistenten Spezifikationseinheiten bzw. Implementations-Varianten verantwortlich sind. Die Beseitigung dieser Inkonsistenzen bedeutet wiederum Modifizieren.

Eine Modifikation wird nach folgender Strategie durchgeführt: Von einer Modifikation direkt betroffene Inkremente der Spezifikation (oder Implementation) werden zunächst im Systemgraph (Modulgraph) mit einer (A)- bzw. (L)-Kennung¹⁾ versehen. Diese Kennung besagt, daß die umhüllende Spezifikationseinheit bzw. Implementations-Variante inkonsistent ist und deshalb bis zur Änderung bzw. Lösung des gekennzeichneten Inkrements für eine weitere Verwendung gesperrt bleibt (unterliegt also der Freigabe-Kontrolle). Nun kann die ursprünglich beabsichtigte Modifikation durchgeführt werden. Somit ist es in einem Entwicklungsteam einem Entwickler möglich, (bei entsprechender Berechtigung) Inkremente der Spezifikation zu modifizieren, ohne daß die Entwickler, die für Systemkomponenten im Wirkbereich der Modifikation zuständig sind, sich ergebende Inkonsistenzen zum Zwecke der Konsistenzerhaltung vorab beseitigen

¹⁾ Formal ist dies ein Knotentyp-/Kantenmarkierungs-Attribut.

müssen. Inkremente mit einem $\textcircled{A}$ bzw. $\textcircled{L}$ werden später dann analog modifiziert.

In den folgenden Abschnitten werden einige interessante inkrementelle Modifikationen und ihre Auswirkungen diskutiert. Wir beschreiben die abstrakte Implementation dieser Inkrementoperationen in Form von Kontrollprozeduren und Graph-Produktionen.

6.2 INKREMENTEINGABE IN EINER SK-SPEZIFIKATION

Typ eines formalen generischen Parameters eines GADT's

Die Situation



Abb. 6-1

ist im Systemgraph durch den Untergraph aus Abb. 6-2 wiedergegeben.

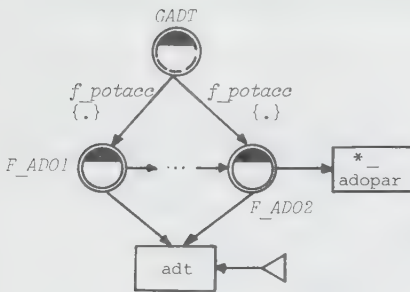
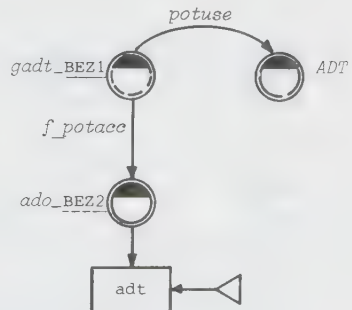


Abb. 6-2

TEST_S.3.3.2E.1 (ADT)¹⁾²⁾

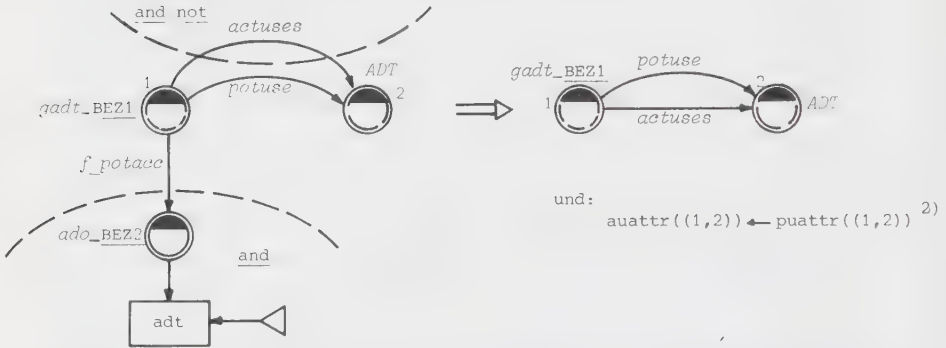


1) Teilgraph-Test.

2) Im weiteren werden ZR- und BR-Attribute wie auch andere Attribute in Tests und Produktionen nur dann berücksichtigt, wenn dies der dargestellte Sachverhalt erfordert.

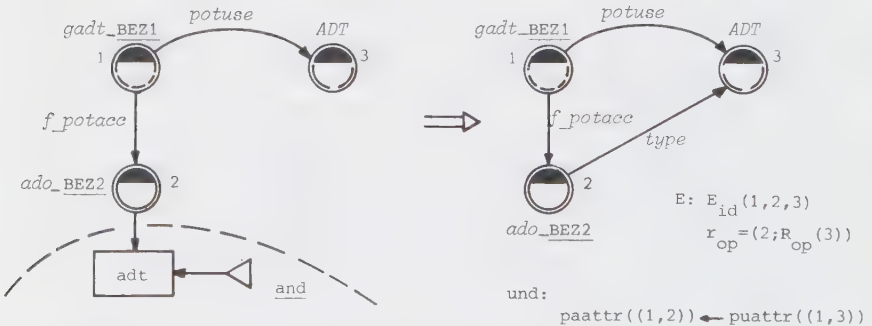
Wird in dieser Situation der Typ (*ADT*) der formalen generischen ADO-Parameter *F ADO_i* eingegeben, testet die Kontrollprozedur *EIN GADT F ADO_{PÄR} TYP* zunächst durch TEST S.3.3.2E.1, ob der aktuell spezifizierte GADT *GADT* explizites Benutzungsrecht bzgl. des ADT's *ADT* besitzt. Bei positivem Testausgang wird vermerkt, daß *ADT* künftig in der äußeren Schnittstelle von *GADT* unter Ausnutzung aller BR-Attribute aktuell benutzt wird.¹⁾

PROD_S.3.3.2E.2 (ADT)



Anschließend wird für jeden ADO-Parameter der Parameterliste die Typbeziehung nachgetragen. Der virtuellen äußeren Schnittstelle jedes ADO-Parameters werden dabei sämtliche als BR-Attribute gegebene Operationen übertragen:

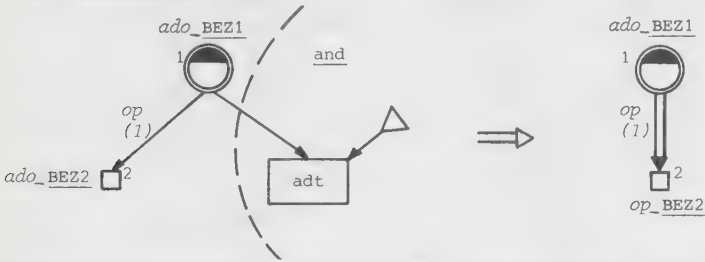
PROD_S.3.3.2E.3 (ADT)



- 1) Der Vermerk wird nur eingetragen, falls er noch nicht existiert. Künftig wird auf die Aktualisierung derartiger technischer Informationen nicht mehr explizit hingewiesen.
- 2) Das Kantenmarkierungs-Attribut *auattr* (*actual used attributes*) beschreibt alle in der vorliegenden Situation aktuell ausgenutzten BR-Attribute. *puattr* hält die *GADT* insgesamt eingeräumten BR-Attribute fest.

Sodann wird der nichtterminale Knoten für den inzwischen eingebrachten ADT gelöscht, nachdem der interne Zeiger jeweils auf die erste Operation jedes ADO-Parameters versetzt ist:

PROD_S.3.3.2E.4



Dem Entwickler werden am Bildschirm nach der Eingabe von ADT sämtliche Operationen angezeigt, die den formalen ADO-Parametern eingeräumt wurden:

ado F_ADO1, ..., F_ADOm : ADT by ▷ OP1, ..., OPn *

Abb. 6-3

Durch selektives Löschen einzelner Operationen läßt sich der Operationensatz dem tatsächlichen Bedarf individuell anpassen (Abmagerung).

Der Typ formaler ADT-Parameter wird analog dem von ADO-Parametern spezifiziert.

Formaler Parameter einer Operation

Gibt ein Entwickler einen zusätzlichen formalen ADO-Parameter ein:

```

op OP ( ...
  :
  ado F_ADO1, ▷ F_ADO■ , F_ADO2, ... * : ADT ;
                adopar-bezeichner
  :
  );
    
```

Abb. 6-4

wird zunächst durch die Kontrollprozedur EIN_OP_F ADOPAR BEZ getestet, ob nicht evtl. bereits ein Parameter mit demselben Bezeichner existiert. Ist dies nicht der Fall, sind sämtliche aktuellen Aufrufe von OP als zu ändernd zu markieren. Diese treten in Implementations-Varianten zu Systemkomponenten auf,¹⁾ die Zugriffsrecht bzgl. eines ADO's mit ZR-Attribut OP besitzen¹⁾:

¹⁾ OP kann Operation eines (G)ADT's sein und an abstrakte Datenobjekte dieses Typs übertragen worden sein.

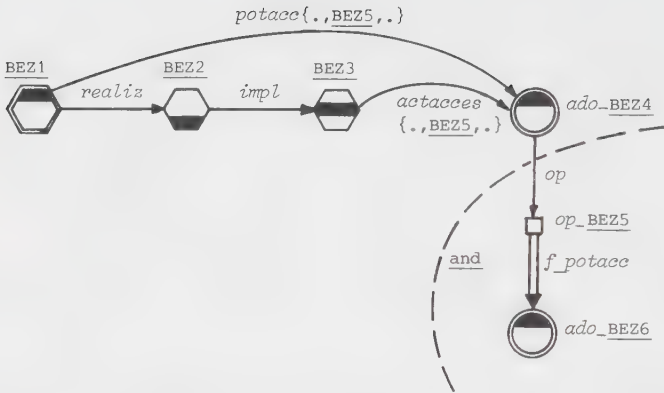


Abb. 6-5

Bereits beim Absetzen des Eingabe-Kommandos war im Systemgraph die Knotenstruktur für einen neuen Parameter in die der Parameterliste eingehängt worden (ähnlich PROD_S.3.5.1E.1 in Abschnitt 4.3.3). Nun ist noch der Bezeichner des neuen Parameters nachzutragen (analog PROD_S.3.5.1E.2).

Die Entwickler, die für die Implementationen von *OP* zuständig sind, sollten eine Mitteilung über den neuen Parameter bekommen.

Die Eingabe eines neuen formalen EDO-Parameters wird auf ähnliche Weise kontrolliert.

Zugriffs-/Benutzungsrechte

Die Expansion des Platzhalters für Zugriffs-/Benutzungsrechte führt zur Ausgabe folgender Rahmenstruktur:

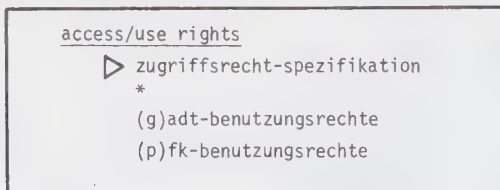
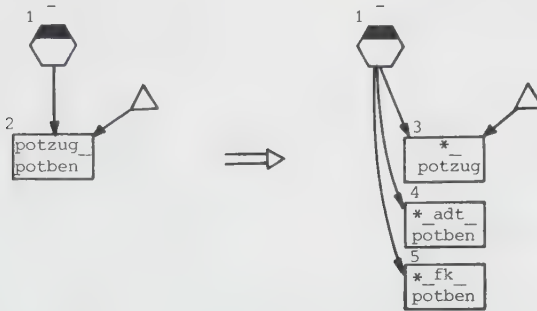


Abb. 6-6

Der Systemgraph wird in *ERZ_POTZUGR_POTBEN* aktualisiert durch:

PROD_S.7E



Zugriffsrecht-Spezifikation

Wurde der Platzhalter für eine Zugriffsrecht-Spezifikation expandiert, kann ein Zugriffsrecht eingetragen werden:

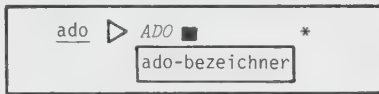
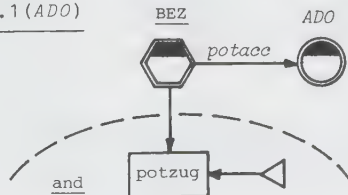


Abb. 6-7

Bevor (in der Kontrollprozedur *EIN POTZUGR SPEZ*) das Zugriffsrecht akzeptiert wird, ist die Korrektheit der Zugriffsrecht-Beziehung zu gewährleisten. Dazu wird zunächst getestet, ob die aktuell spezifizierte Systemkomponente *SK* bereits ein Zugriffsrecht bzgl. *ADO* besitzt:

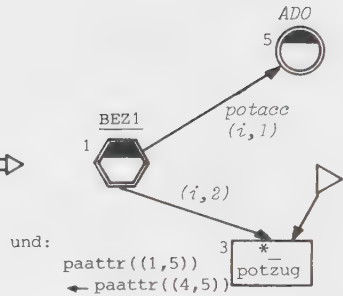
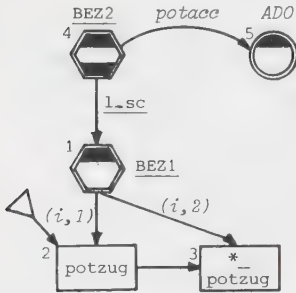
TEST_S.7.1.1E.1(ADO)



Anschließend trägt *EIN POTZUGR SPEZ SK* ein Zugriffsrecht ein, falls *ADO* sich im Zugriffsbereich befindet.

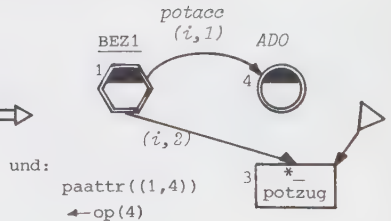
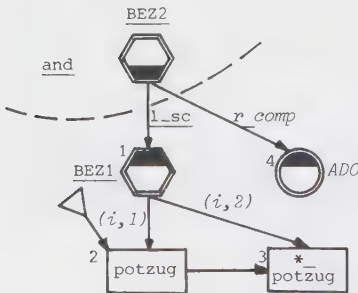
PROD S7.1.1E.2 übernimmt dies für den Fall, daß die Vater-Systemkomponente explizites Zugriffsrecht bzgl. *ADO* besitzt. Sämtliche ZR-Attribute dieser Zugriffsrecht-Beziehung sind zunächst auch ZR-Attribute der neuen Beziehung:

PROD_S.7.1.1E.2 (ADO)



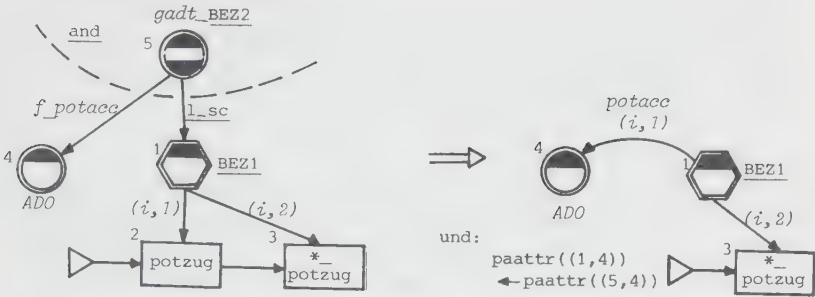
Falls die Vater-Systemkomponente implizites Zugriffsrecht bzgl. *ADO* besitzt und *ADO* seinerseits nicht direkt oder indirekt Zugriffsrecht bzgl. der aktuell spezifizierten Systemkomponente besitzt, wird *SK* ein Zugriffsrecht bzgl. *ADO* mit sämtlichen *ADO*-Operationen als ZR-Attributen eingeräumt:

PROD_S.7.1.1E.3 (ADO)



Könnte keine der beiden Produktionen angewendet werden, könnte *ADO* noch formaler generischer ADO-Parameter sein, falls die Vater-Systemkomponente ein GADT ist:

PROD_S.7.1.1E.4 (ADO)



Liegt auch diese Situation nicht vor, ist das Zugriffsrecht trotzdem zulässig, falls das ZR-Subjekt Wurzel eines Teilsystems ist und das ZR-Objekt in einem anderen, noch nicht integrierten Teilsystem liegt. Dieser Test wird im Zuge einer Benutzungsrecht-Spezifikation diskutiert werden.

Konnte das Zugriffsrecht eingetragen werden, werden dem Entwickler sämtliche übertragenen ZR-Attribute angezeigt:

ado ADO ▷ * by OP1, OP2, ..., OPn;

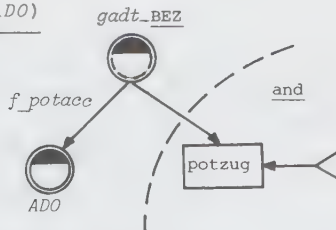
Abb. 6-8

Dieser Satz an ZR-Attributen kann nun bei Bedarf individuell abgemagert werden.

Erwies sich das einzutragende Zugriffsrecht als unzulässig, kann für gezieltere Fehlermeldungen noch geprüft werden, ob eine der folgenden Situationen vorliegt:

- ADO ist kein ADO (trivialer Test)
- ADO ist mit dem ZR-Subjekt identisch (trivialer Test)
- ADO ist formaler generischer ADO-Parameter, falls das ZR-Subjekt ein GADT ist¹⁾:

TEST_S.7.1.1E.7(ADO)



¹⁾ Analog wird im Falle, daß das ZR-Subjekt eine PFK ist, überprüft, ob nicht ein formaler ZR-Parameter ADO existiert.

Sind bereits vor der Eingabe des neuen Zugriffsrechts Zugriffsrechte in der Liste enthalten:

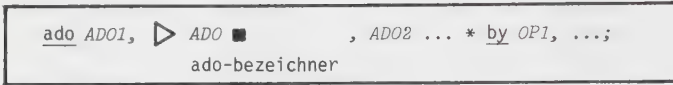
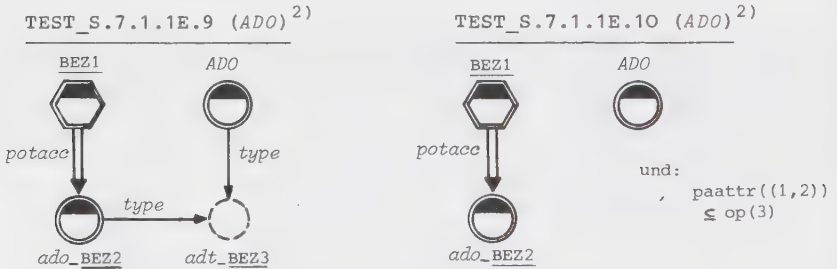


Abb. 6-9

sind zusätzliche Tests nötig.

Enthält die Zugriffsrecht-Liste zunächst nur ein Zugriffsrecht, ist abzusichern, daß das entsprechende ZR-Objekt mittels explizitem Typ deklariert wurde¹⁾ (trivialer Test). Bei positivem Testausgang bzw. im Falle einer Liste mit mehreren Zugriffsrechten ist (neben dem Test auf Doppelspezifikation) zu prüfen, ob die bisherigen ZR-Objekte und *ADO* vom selben expliziten Typ sind (TEST_S.7.1.1E.9).



Anschließend ist zu sichern, daß jedes ZR-Attribut der bisherigen ZR-Objekte auch durch eine *ADO*-Operation abgedeckt ist (TEST_S.7.1.1E.10).

PROD_S.7.1.1E.11 - 13 tragen dann die Zugriffsrecht-Beziehung ein, falls *ADO* im Zugriffsbereich liegt. Diese Produktionen ähneln PROD_S.7.1.1E.2 - 4, wobei jedoch die aktuelle Position des internen Zeigers der in TEST_S.7.1.1E.9 entspricht.

Die Entwickler, die für die Realisierungs- und Implementations-Varianten der aktuell spezifizierten Systemkomponente verantwortlich sind, sollten vom neuen Zugriffsrecht in Form einer Meldung Kenntnis erhalten.

Benutzungsrecht-Spezifikation

Bei Eingabe eines Benutzungsrechts bzgl. eines (G)ADT's (G)ADT

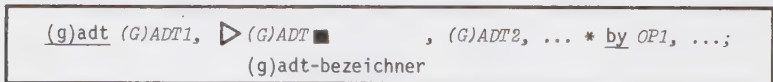


Abb. 6-10

¹⁾ ZR-Objekte in einer Zugriffsrecht-Liste müssen typäquivalent sein.

²⁾ Für den Fall, daß der interne Zeiger auf dem *-Platzhalter-Knoten der Zugriffsrecht-Liste steht, ist ein ähnlicher Test vorzusehen.

ist zunächst abzusichern, daß die aktuell spezifizierte Systemkomponente (BR-Subjekt) nicht bereits ein Benutzungsrecht bzgl. (G)ADT (G)ADT besitzt (einfacher Test).

Anschließend wird geprüft, ob durch das neue Benutzungsrecht kein Zyklus in der um die Benutzungsrecht-Beziehung erweiterten Realisiert-Beziehung entsteht (Wahrung von KB_NICHTREK_POTBEN). Dazu wird zunächst die Identität von BR-Subjekt und BR-Objekt ((G)ADT) getestet (trivialer Test). Liegt keine Identität vor, wird zu sämtlichen Systemkomponenten, die von (G)ADT aus in der erweiterten Realisiert-Beziehung¹⁾ erreichbar sind, eine *potuse**-Kante als Hilfskante gezogen:

PROD_S.7.2.1E.3 ((G)ADT)



KB_NICHTREK_POTBEN ist verletzt, wenn eine *potuse**-Kante von (G)ADT zum BR-Subjekt existiert (einfacher Test). Das neue Benutzungsrecht wird zurückgewiesen. In jedem Fall sind alle *potuse**-Kanten wieder zu löschen (triviale Produktion).

Anschließend an die Nichtrekursivität ist die Gewährleistung des Prinzips "information hiding" (KB_INFID_POTBEN) zu überprüfen: Das BR-Subjekt muß der Teilhierarchie der Realisierungs-Variante angehören, die das BR-Objekt als lokalen (G)ADT einführt. Um dies festzustellen, werden *glob**-Hilfskanten vom Knoten des BR-Subjekts zu allen Realisierungs-Varianten gezogen, die sich durch Aufwärtsverfolgen der Systemhierarchie erreichen lassen:

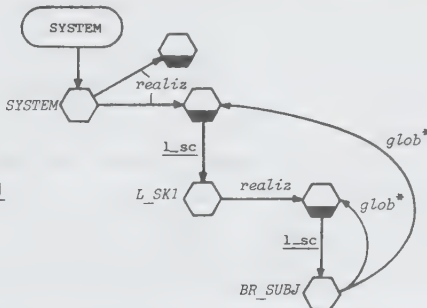
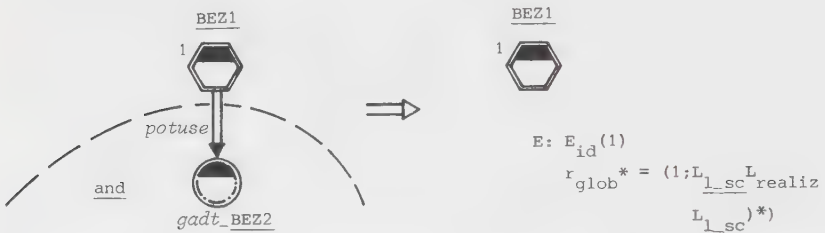


Abb. 6-11

¹⁾ Es ist zu berücksichtigen, daß (G)ADT evtl. eine Typausprägung ist (siehe Abschnitt 3.3.3).

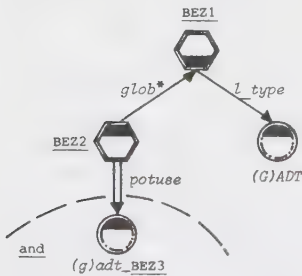
Die $glob^*$ -Kanten werden eingezogen durch:

PROD_S.7.2.1E.6¹⁾

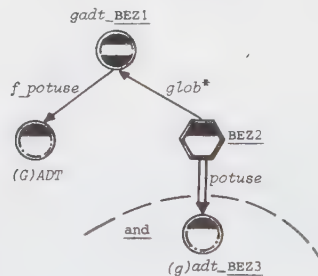


Liegt anschließend die Situation aus TEST.7.2.1E.8 (BR-Subjekt und (G)ADT besitzen eine "überdeckende" Realisierungs-Variante) oder aus TEST S.7.2.1E9 (BR-Subjekt ist lokal in einem GADT mit (G)ADT als formalem ADT-Parameter) vor, ist KB_INFHD_POTBEN gewährt.

TEST_S.7.2.1E.8 ((G)ADT)



TEST_S.7.2.1E.9 ((G)ADT)



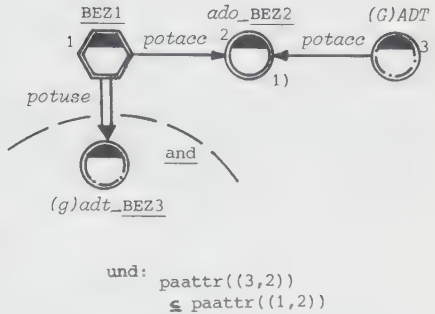
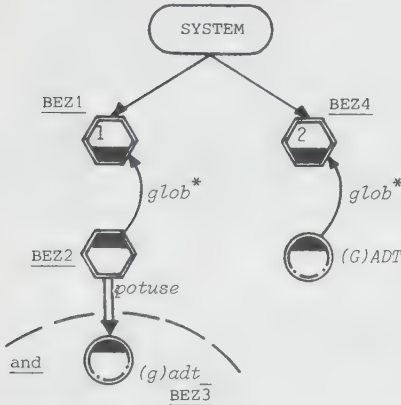
Ergibt sich keine der beiden Situationen, ist das Prinzip "information hiding" nur dann sicher verletzt, wenn sich BR-Subjekt und BR-Objekt nicht in zwei Teilsystemen befinden, die noch nicht integriert sind.²⁾ Um dies zu prüfen, werden (analog PROD_S.7.2.1E.6) zusätzlich $glob^*$ -Hilfskanten gezogen, die vom Knoten des BR-Objekts (G)ADT ausgehen. Liegt anschließend die Situation aus TEST 7.2.1E.10 vor, befinden sich BR-Subjekt und BR-Objekt in zwei verschiedenen Teilsystemen. Bei negativem Testausgang wird endgültig gegen das Prinzip des information hidings verstoßen. In jedem Fall sind alle $glob^*$ -Kanten wieder aus dem Systemgraph zu nehmen (triviale Produktion).



2) Die Überprüfung der Konsistenz wird dann bis zur Integration zurückgestellt.

TEST_S.7.2.1E.10 ((G)ADT)

TEST_S.7.2.1E.12 ((G)ADT)



Neben dem Prinzip "information hiding" muß auch die Verträglichkeit mit der Vererbungsregel (KB_VERERBVERTR_POTBEN) garantiert sein. Diese ist nur dann gegeben, wenn das BR-Subjekt mindestens alle Zugriffsrechte des BR-Objekts besitzt und auch die jeweiligen ZR-Attribute die entsprechenden des BR-Objekts überdecken. Für alle Zugriffsrechte des BR-Objekts muß also der TEST_S.7.2.1E.12 positiv verlaufen.

Das neue Benutzungsrecht wird endgültig akzeptiert, wenn das BR-Objekt auch sämtliche Operationen bereithält, die als BR-Attribute vorgegeben sind.

Das neue Benutzungsrecht ist den Entwicklern der Realisierungs- und Implementations-Varianten zur aktuell spezifizierten Systemkomponente mitzuteilen.

6.3 INKREMENTÄNDERUNG UND -LÖSCHUNG IN EINER SK-SPEZIFIKATION

Das Löschen eines komplexen Inkrements löst das Löschen seiner lokalen Inkremente aus. Deshalb sind neben Änderungsoperationen nur Löschoperationen atomarer Inkremente zu diskutieren.

Bezeichner einer Systemkomponente

Der Austausch des Bezeichners einer Systemkomponente macht nur einen Test auf mögliche Doppeldeklaration (ähnlich TEST_S.3.1E.1/2) und die Aktualisierung des Systemgraphen nötig. Angewandte Auftreten der betroffenen Systemkomponente sind - aufgrund der im Systemgraph zentral gehaltenen Information - von der Änderung nicht

1) Das ZR-Objekt kann natürlich auch ADO-Parameter eines GADT's sein.

berührt.

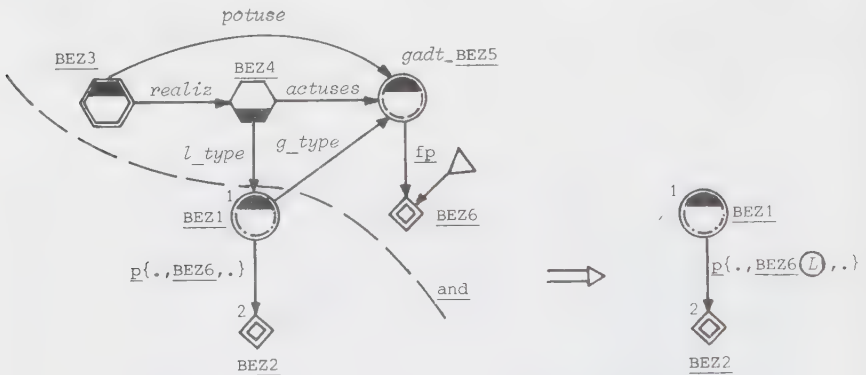
Der Bezeichner einer Systemkomponente kann nicht gelöscht werden.

Formaler generischer Parameter eines GADT's

Das für die Änderung des Bezeichners einer Systemkomponente Gesagte gilt auch für den Austausch des Bezeichners eines formalen generischen Parameters.

Wird ein formaler generischer Parameter F_PAR eines GADT's $GADT$ ¹⁾ gelöscht, sind davon zunächst sämtliche aktuellen $GADT$ -Benutzungen ¹⁾ in Realisierungs-Varianten zu Systemkomponenten betroffen, die bzgl. $GADT$ explizites/implizites Benutzungsrecht besitzen. Für den Aktualparameter, der F_PAR entspricht, ist ein Lösch-Vermerk einzubringen:

PROD_S.3.5.1L.1²⁾



PROD_S.3.5.1L.1 behandelt den Fall des expliziten Benutzungsrechts bzgl. $GADT$; für den entsprechenden Fall des impliziten Benutzungsrechts ist eine ähnliche Produktion PROD_S.3.5.1L.2 vorzusehen.

Nun sind noch alle angewandten Auftreten des zu löschenden Formalparameters selbst zu ändern.

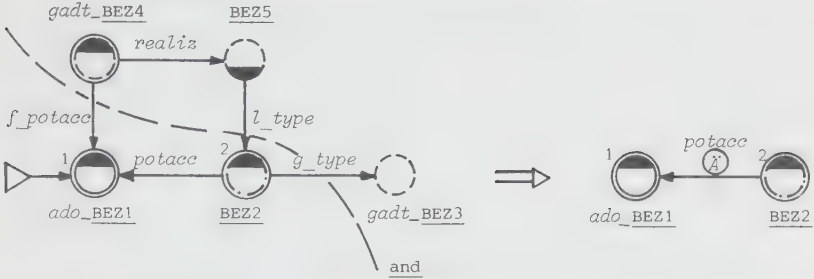
Ist F_PAR ein ADO-Parameter, kann er als Aktualparameter in Typkonkretisierungen in $GADT$ -Realisierungs-Varianten auftreten ³⁾. Diese aktuellen Zugriffsrechte sind mit einem Änderungsvermerk zu versehen:

1) Dienen ausschließlich zu Typkonkretisierungen.

2)  repräsentiert einen Parameter beliebiger Art, entspr. Aktualparameter.  den

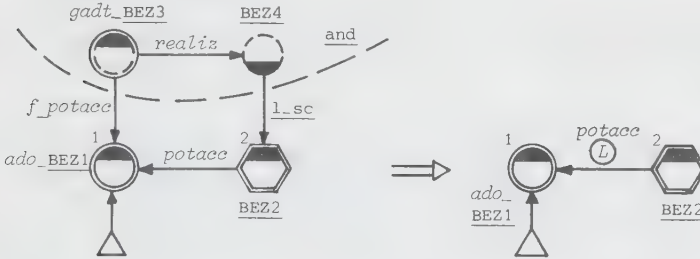
3) Ebenso als aktuelles Zugriffsrecht in Benutzungsspezifikationen.

PROD_S.3.5.1L.3



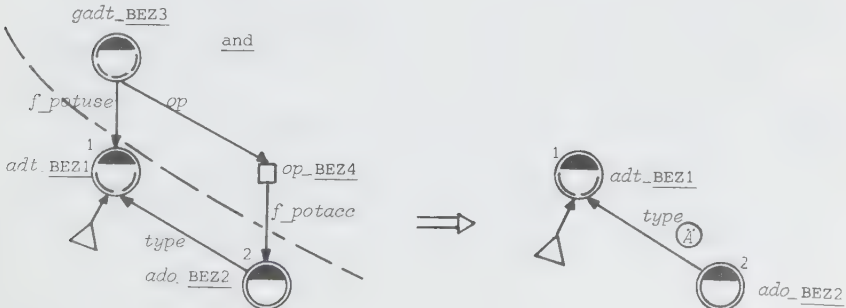
Alle aktuellen Zugriffe auf F_{PAR} in GADT-Implementations-Varianten sind als zu löschend zu markieren.
 F_{PAR} kann auch ZR-Objekt einer Sohn-Systemkomponente in einer beliebigen GADT-Realisierungs-Variante sein. Derartige Beziehungen erhalten einen Lösch-Vermerk:

PROD_S.3.5.1L.4



Ist der zu löschende Formalparameter F_{PAR} ein ADT-Parameter, könnten zunächst GADT-Operationen einen formalen ADO-Parameter vom Typ F_{PAR} besitzen. Der Typ dieser formalen ADO-Parameter ist auszutauschen.

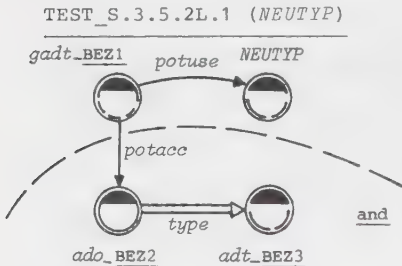
PROD_S.3.5.1L.5



Ebenso ist in *GADT*-Realisierungs-Varianten, die *F_{PAR}* aktuell benutzen, jedes angewandte Auftreten von *F_{PAR}* mit einem Austausch-Vermerk zu versehen.¹⁾ Auch in Typkonkretisierungen in *GADT*-Realisierungs-Varianten, in denen *F_{PAR}* als aktuelles BR-Objekt an einen (G)ADT durchgereicht wird, ist *F_{PAR}* auszutauschen. Schließlich ist jedes Benutzungsrecht, das einer Systemkomponente bzgl. *F_{PAR}* eingeräumt wurde, als "zu löschend" zu markieren (Produktion ähnlich PROD_S.3.5.1L.4). Nach diesen konsistenznerhaltenden Aktionen wird *F_{PAR}* gelöscht.²⁾

Typ eines formalen generischen Parameters eines GADT's

Wird der Typ von formalen generischen (ADO-/ADT-)Parametern eines GADT's *GADT*, die eine Liste bilden, ausgetauscht (alleiniges Löschen ist unzulässig), ist zunächst zu testen, ob bzgl. des neuen Typs *NEUTYP* ein Benutzungsrecht vorliegt, z.B. bei formalen ADO-Parametern durch TEST_S.3.5.2L.1. Bei positivem Testausgang sind in sämtlichen aktuellen Benutzungen von *GADT* alle Aktualparameter, die Formalparameter vom zu ändernden Typ absättigen, mit einem Änderungsvermerk zu versehen (ähnlich Produktionen PROD_S.3.5.1L.1/2). Derartige aktuelle Benutzungen treten in Realisierungs-Varianten zu Systemkomponenten auf, die bzgl. *GADT* explizites/implizites Benutzungsrecht besitzen.



Nun sind noch die aktuellen Auftreten aller vom Typ-Austausch betroffenen Formalparameter konsistenznerhaltenden Maßnahmen zu unterziehen. Sei *F_{PAR}* ein solcher Formalparameter. Handelt es sich bei *F_{PAR}* um einen ADO-Parameter, sind in den *GADT*-Realisierungs-Varianten sämtliche Auftreten als aktuelles Zugriffsrecht mit einem Änderungsvermerk zu versehen (siehe PROD_S.3.5.1L.3). Das gleiche gilt für aktuelle Zugriffe bzgl. *F_{PAR}* in Implementations-Varianten. Ist *F_{PAR}* für eine Sohn-Systemkomponente in einer *GADT*-Realisierungs-Variante ZR-Objekt, erhalten sämtliche ZR-Attribute einen Lösch-Vermerk³⁾, da der neue Typ *NEUTYP* andere Operationen bereitstellt. Ist *F_{PAR}* ein ADT-Parameter, sind zunächst die aktuellen Benutzungen in *GADT*-Realisierungs-Varianten konsistent zu halten, in denen die Operationen von *F_{PAR}* eine Rolle spielen. Dies betrifft zum einen Deklarationen, in denen abstrakte Datenobjekte vom Typ *F_{PAR}* eingeführt werden: Sämtliche diesen abstrakten Datenobjekten übertragene Operationen sind als "zu löschend" zu markieren. Gleiches gilt zum anderen für alle formalen ZR-Objekte eines GADT's (bzw.

- 1) Sämtliche möglichen angewandten Auftreten wurden in Abschn. 2.2.4.4 aufgelistet.
- 2) Es wird nur die Kante des formalen Zugriffs-/Benutzungsrechts und nicht der Knoten zu *F_{PAR}* gelöscht: Bzgl. *F_{PAR}* bestehen noch Bezüge mit einer $\textcircled{A}$ - bzw. $\textcircled{L}$ -Kennung.
- 3) Die Zugriffsrecht-Beziehung als solche sollte weiterexistieren.

einer PFK), falls sie vom Typ F_PAR sind.

Außerdem erhalten sämtliche Auftreten von F_PAR als aktuelle Zugriffsrechte in Typkonkretisierungen einen Änderungsvermerk, da sie die Spezifikation des entsprechenden formalen Zugriffsrechts aufgrund des sich ändernden Operationensatzes nicht mehr absättigen.

Dieselben konsistenznerhaltenden Maßnahmen sind für angewandte Auftreten von F_PAR in Systemkomponenten zu ergreifen, die explizites Benutzungsrecht bzgl. F_PAR besitzen.

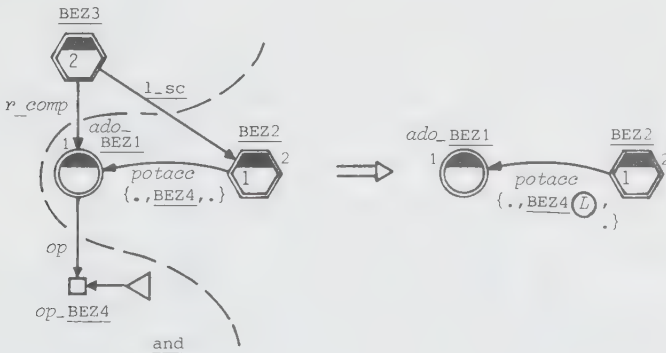
Im Anschluß an diese konsistenznerhaltenden Aktionen wird im Systemgraph die Typbeziehung der generischen Formalparameter zum zu ändernden Typ gelöst, indem ein entsprechender Platzhalter-Knoten dessen Stelle einnimmt.

Die Typbeziehung der Formalparameter zu $NEUTYP$ wird schließlich durch die Produktionen eingebracht, die für die Eingabe eines Parameter-Typs zuständig sind.¹⁾

Spezifikation einer Operation/Funktion

Bevor eine Operation OP zu einem (mittels implizitem ADT) deklarierten ADO ADO gelöscht werden kann, ist in Bruder-Systemkomponenten zu ADO , die ein Zugriffsrecht bzgl. ADO mit ZR-Attribut OP besitzen, dieses ZR-Attribut als "zu löschend" zu markieren:

PROD_S.6.1L.1



Anschließend wird OP aus der äußeren Schnittstelle gelöscht (ähnlich PROD_S.3.3.3L.4).

Handelt es sich bei OP um die Operation eines (G)ADT's ADT bzw. $GADT$, ist zunächst die Konsistenz aller angewandten Auftreten des (G)ADT's zu wahren:

- Wurde einem ADO ADO vom Typ ADT OP übertragen, ist diese ADO -Operation als "zu löschend" zu markieren (dies gilt analog für einen ADT vom Typ $GADT$).

¹⁾ siehe Abschnitt 4.3.3

- *Fungiert ADT als aktuelles Benutzungsrecht, ist zu prüfen, ob das korrespondierende formale Benutzungsrecht das BR-Attribut OP besitzt. In diesem Fall ist das aktuelle Benutzungsrecht zu ändern.*

Die aufgeführten Maßnahmen zur Konsistenzwahrung sind für die Realisierungs-Variante, die *ADT/GADT* einführt, durchzuführen. Für alle Systemkomponenten, die bzgl. *ADT/GADT* explizites Benutzungsrecht besitzen, ist das BR-Attribut *OP* mit einem Lösch-Vermerk zu versehen. Das Löschen von *OP* wird wieder ähnlich *PROD_S.3.3.3L.4* durchgeführt.

Das Löschen einer Funktion wird ähnlich vollzogen.

Bezeichner einer Operation/Funktion

Ein Operationen-/Funktionen-Bezeichner kann gegen einen neuen ausgetauscht werden, falls keine Doppeldeklaration vorliegt (triviale Produktion).

Formalparameter einer Operation/Funktion

Der Austausch eines Parameter-Bezeichners kann nach Test auf evtl. Doppeldeklaration vollzogen werden.

Das Löschen eines formalen Parameters einer Operation (Funktion) *OP* erfordert das Löschen der entsprechenden aktuellen Parameter in aktuellen Aufrufen von *OP*. Davon betroffen sind alle Implementations-Varianten zu Systemkomponenten, die Zugriffs- bzw. Benutzungsrecht bzgl. eines ZR-(bzw. BR-)Objekts besitzen, wobei *OP* ZR-(bzw. BR-)Attribut ist. Zu löschen ist auch jedes angewandte Auftreten des zu löschenden formalen Parameters in allen Implementationen von *OP*. Die anschließende Aktualisierung des Systemgraphen erfolgt über eine einfache Produktion.

Typ eines Formalparameters einer Operation/Funktion

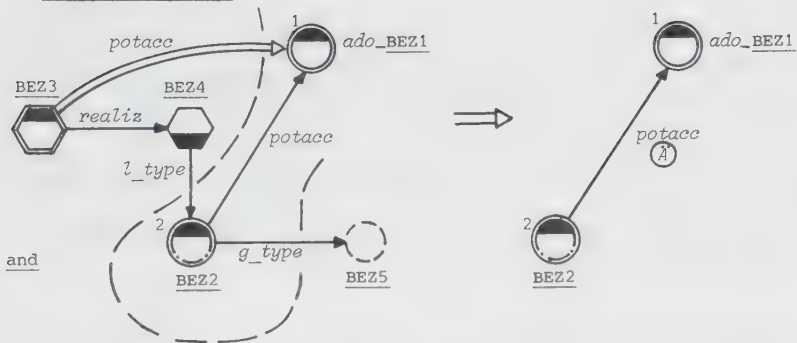
Der Austausch des Typs eines Formalparameters erfordert zunächst die Überprüfung des neuen Typs hinsichtlich seiner Benutzbarkeit (vgl. "Typ eines formalen generischen Parameters eines GADT's"). Die anschließenden konsistenzerhaltenden Maßnahmen ähneln denen, die beim Löschen eines Formalparameters nötig sind. Anstelle von Löschungen sind dabei jedoch nur Austauschen nötig. Bevor schließlich der neue Typ eingegeben wird, wird im Systemgraph die ursprüngliche Typbeziehung gelöst und dafür der entsprechende Platzhalter-Knoten eingebracht.

Zugriffsrecht

Das Löschen¹⁾ eines Zugriffsrechts bzgl. *ADO ADO* bewirkt, daß in jeder Realisierungs-Variante des ZR-Subjekts, die dieses Zugriffsrecht an lokale Systemkomponenten als aktuelles Zugriffsrecht weiterreicht, dieses aktuelle Zugriffsrecht einen Austausch-Vermerk erhält:

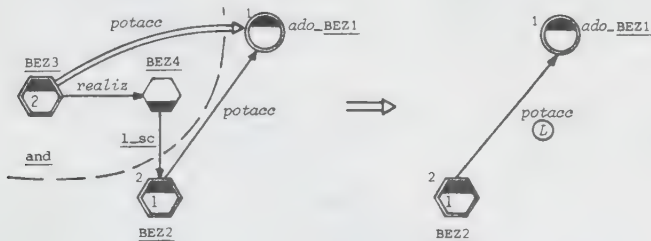
¹⁾ Ein "Ändern" ist durch Löschen und Neu-Eingabe zu erreichen.

PROD_S.7.1.1L.1



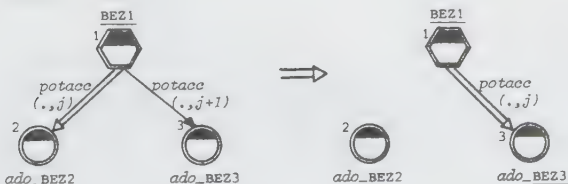
(Falls ADO aktuelles ZR-Objekt in einer Benutzungsspezifikation ist, ist eine ähnliche Produktion (PROD_S.7.1.1L.2) anzuwenden.
In allen Implementations-Varianten des ZR-Subjekts sind sämtliche aktuellen Zugriffe mit einem Löschraster zu versehen.
In allen lokalen Systemkomponenten, die das Zugriffsrecht bzgl. ADO vererbt bekommen haben, ist dieses Zugriffsrecht als "zu löschen" zu markieren:

PROD_S.7.1.1L.3



Anschließend erfolgt das Löschen des Zugriffsrechts:

PROD_S.7.1.1L.4¹⁾



¹⁾ Für den Fall, daß der interne Zeiger auf den *-Platzhalter-Knoten der Liste von Zugriffsrechten deutet, ist eine weitere Produktion vorzusehen.

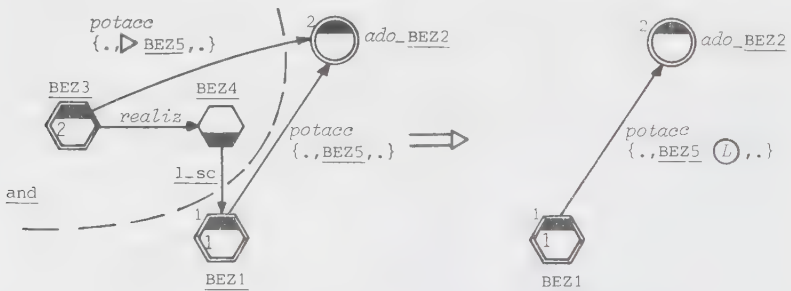
Zugriffsrecht-Attribut

Soll das ZR-Attribut *OP* eines Zugriffsrechts gelöscht werden, das bzgl. eines ADO's *ADO* eingeräumt wurde, sind davon zunächst sämtliche Situationen betroffen, in denen das Zugriffsrecht an lokale Systemkomponenten als aktuelles Zugriffsrecht weitergereicht wurde: Besitzt das entsprechende formale Zugriffsrecht *OP* als ZR-Attribut, ist das aktuelle Zugriffsrecht als "auszutauschend" zu markieren (Produktion ähnlich PROD_S.7.1.1L.1/2).

In den Implementations-Varianten des ZR-Subjekts erhalten sämtliche aktuellen Zugriffe auf das ZR-Objekt mittels *OP* einen Lösch-Vermerk.

In allen lokalen Systemkomponenten, die das Zugriffsrecht bzgl. *ADO* mit dem ZR-Attribut *OP* vererbt bekommen haben, ist dieses ZR-Attribut mit einem Lösch-Vermerk zu versehen:

PROD_S.7.1.2L.3



Anschließend wird das ZR-Attribut *OP* gelöscht durch:

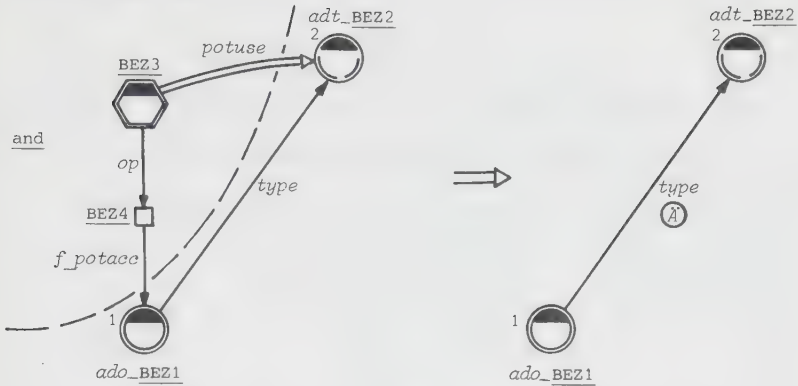
PROD_S.7.1.2L.4



Benutzungsrecht

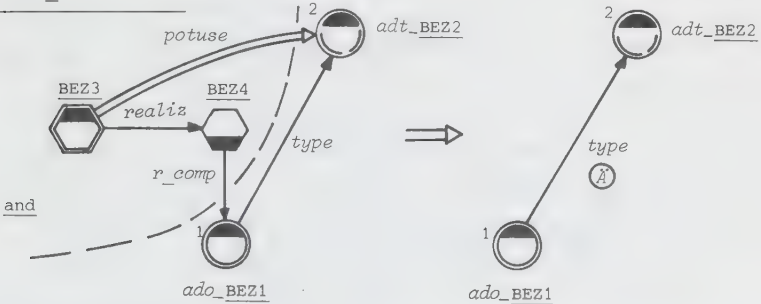
Vor dem Löschen eines Benutzungsrechts, das der Systemkomponente *SK* bzgl. eines ADT's *ADT* eingeräumt ist, sind für die Spezifikationen der *SK*-Operationen/-Funktionen evtl. konsistenznerhaltende Maßnahmen nötig: Jeder formale ADO-Parameter vom Typ *ADT* ist hinsichtlich seines Typs zu ändern:

PROD_S.7.2.1L.1



Ist das BR-Subjekt ein GADT, könnte *ADO* auch der Typ formaler generischer ADO-Parameter sein. Diese angewandten Auftreten sind mit einer Änderungskennung zu versehen. Sättigt *ADT* in den Realisierungs-Varianten des BR-Subjekts formale Benutzungsrechte in Typkonkretisierungen ab, so sind diese aktuellen Benutzungsrechte auszutauschen (Produktion ähnlich PROD_S.7.1.1L.1). Ebenso ist jedes angewandte Auftreten von *ADT* in den Realisierungs-Varianten mit einem Änderungs-Vermerk zu markieren, in denen *ADT* der Typ eines lokalen ADO's ist:

PROD_S.7.2.1L.3



Ist das Benutzungsrecht bzgl. eines GADT's *GADT* zu löschen, ist der Typ formaler generischer ADT-Parameter vom Typ *GADT* zu ändern, falls das BR-Subjekt ein GADT ist. Außerdem ist der Typ der lokalen abstrakten Datentypen¹⁾ zu ändern, die vom generischen Typ *GADT* sind (ähnlich PROD_S.7.2.1L.3).

¹⁾ Evtl. "Phantasie"-ADT

Anschließend an die konsistenzverhaltenden Maßnahmen wird das Benutzungsrecht bzgl. *ADT* bzw. *GADT* gelöscht (ähnlich *PROD_S.7.1.1L.4*).

6.4. INKREMENTEINGABE IN EINER REALISIERUNGS-VARIANTE

Das Kommando zur Eingabe einer Realisierungs-Variante bewirkt die Bildschirm-Ausgabe

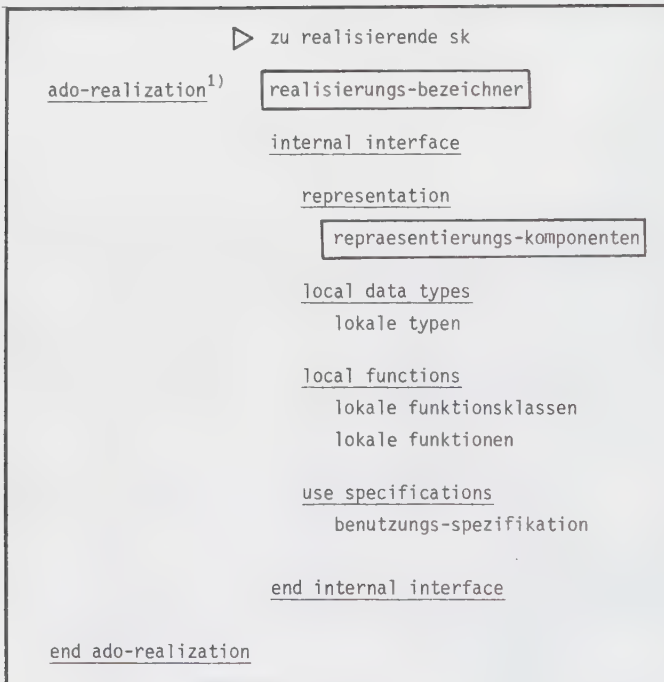
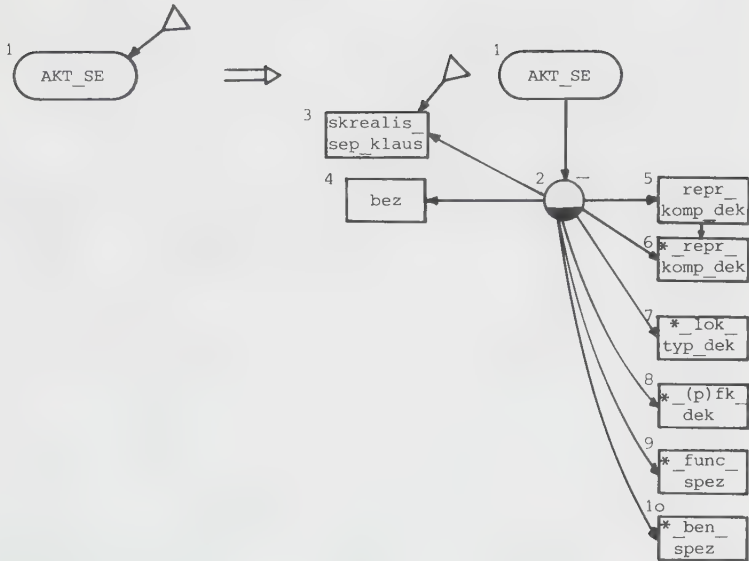


Abb. 6-12

Der Systemgraph wird in der Kontrollprozedur *ERZ_ADO_REALIS_RUMPF* durch *PROD_RV.1E* aktualisiert:

¹⁾ In diesem Beispiel wird die Realisierungs-Variante eines ADO's spezifiziert.

PROD_RV.1E



Repräsentierungs-Komponenten

Zur Eingabe der Deklaration von Repräsentierungs-Komponenten wird das Menue

Alternativen: ☒ ado-dekl. mittels impl. adt ☐ ado-dekl. mittels expl. adt ☐ ado-dekl. mittels gadt ☐ edo-dekl. ☐ dynamische Repräsentierung

Abb. 6-13

ausgegeben. Je nach gewählter Alternative wird im Systemgraph die zugehörige Listenstruktur initialisiert.

ADO-Deklaration mittels explizitem ADT

Ist der Bezeichner eines ADO's in einer ADO-Deklaration mittels explizitem ADT eingebracht:

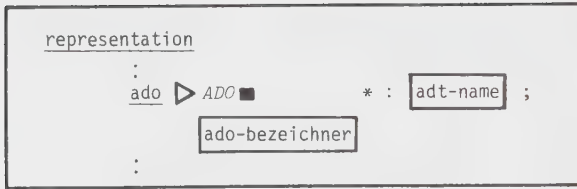
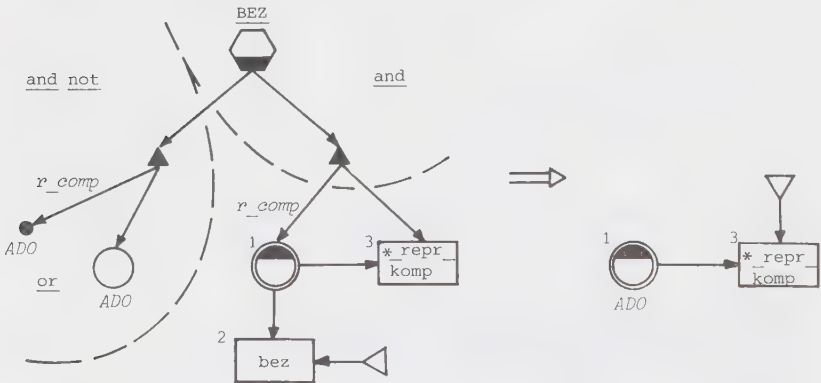


Abb. 6-14

aktualisiert die Kontrollprozedur *EIN_ADO_DEKL_M_EXPL_ADT* den Systemgraph, falls es noch keine Repräsentierungs-Komponente *ADO* gibt, durch:

PROD_RV.5.3.1E.1 (ADO)



Existiert in der gleichen Liste bereits mindestens eine Repräsentierungs-Komponente, ist bei der Deklaration eines weiteren ADO's *ADO* die Eingabe ähnlich PROD_S.3.5.1E.1 (Eingabe eines generischen Formalparameters) vorzubereiten. Die Abspeicherung des eingegebenen Bezeichners erfolgt dann wieder durch PROD_RV.5.3.1E.1.

Typ eines abstrakten Datenobjekts

Wird der Typ von abstrakten Datenobjekten spezifiziert, die in einer Liste deklariert werden:

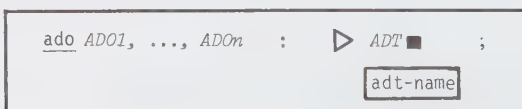


Abb. 6-15

liegt im Systemgraph vorher die Situation von Abb. 6-16 vor. Die Kontrollprozedur *EIN_ADO_DEKL_M_EXPL_ADT_TYP* prüft zunächst, ob die Systemkomponente zur aktuell spezifizierten Realisierungs-Variante Benutzungsrecht bzgl. *ADT* besitzt. Trifft eine von drei zulässigen Konstellationen zu, wird die aktuelle Benutzung von *ADT* vermerkt. Ist das Benutzungsrecht bzgl. *ADT* explizit, übernimmt *PROD_RV.5.3.2E.1* diesen Eintrag.

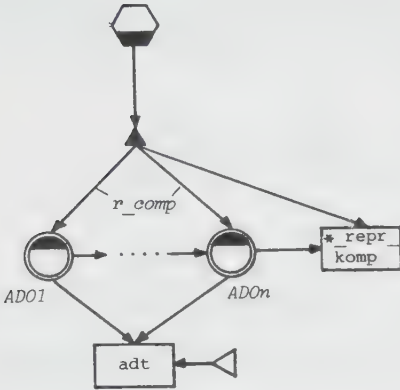
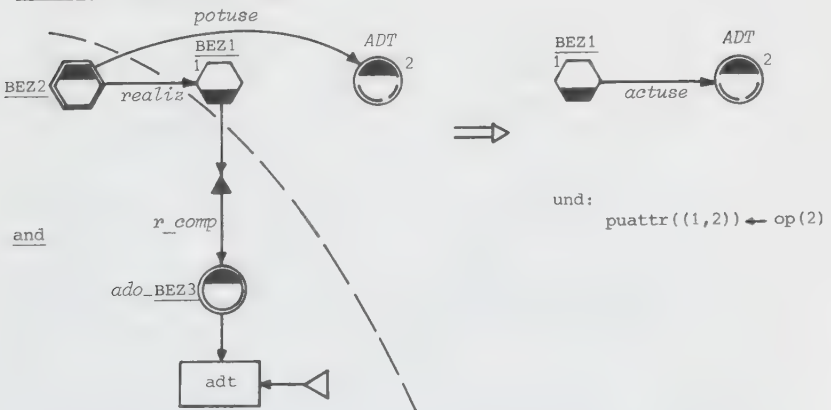


Abb. 6-16

*PROD_RV.5.3.2E.1 (ADT)*¹⁾



Ähnlich diesem Fall eines expliziten Benutzungsrechts sind Produktionen *PROD_RV.5.3.2E.2* und *PROD_RV.5.3.2E.3* anzuwenden, falls *ADT* formaler generischer *ADT*-Parameter der Vater-Systemkomponente oder eine Bruder-Systemkomponente ist.

¹⁾ Zwischen den mit *BEZ1* und *ADT* markierten Knoten gibt es bisher keine *actuse*-Kante (Untergraphen-Test).

PROD_RV.5.3.2E.4 stellt (ähnlich PROD_S.3.3.2E.3) für jedes in der Liste deklarierte ADO die Typbeziehung her und räumt diesem ADO sämtliche benutzbaren ADT-Operationen ein. Gleichzeitig wird dem Entwickler dieser Operationensatz zur eventuellen Abmagerung angeboten (vgl. Eingabe des Typs formaler generischer ADO-Parameter in Abschn. 6.2).

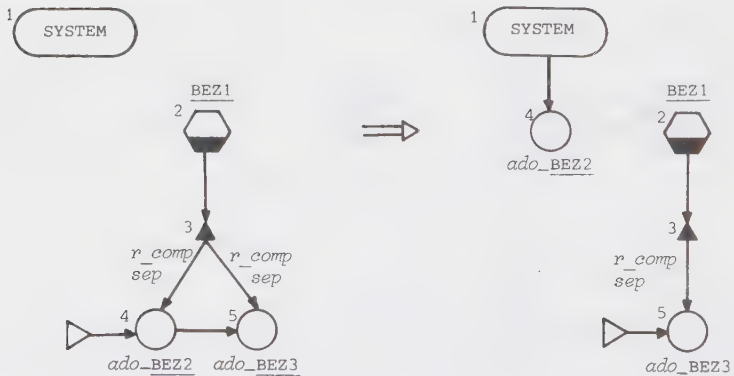
6.5 INKREMENTÄNDERUNG UND -LÖSCHUNG IN EINER REALISIERUNGS-VARIANTE

Repräsentierungs-Komponente

Der Bezeichner einer Repräsentierungs-Komponente kann nach einem Test auf Doppeldeklaration ohne zusätzliche konsistenzverhaltende Maßnahmen ausgetauscht werden.

Das Löschen des Verweises auf ein getrennt spezifiziertes ADO bewirkt das Lösen der Realisiert-Beziehung zwischen ADO und seiner Vater-Systemkomponente:

PROD_RV.5.2.1L.1



Zuvor sind in den Implementations-Varianten sämtliche aktuellen Zugriffe auf ADO und in derselben Realisierungs-Variante alle Zugriffsrechte bzgl. ADO mit einem Lösch-Vermerk zu versehen. Beim Löschen eines ADO's ADO, der mittels eines expliziten ADT's deklariert ist, ist zunächst jedes Auftreten von ADO als aktuelles ZR-Objekt in derselben Realisierungs-Variante als zu ändernd zu markieren. Zusätzlich sind dieselben konsistenzverhaltenden Maßnahmen wie im Falle eines getrennt spezifizierten ADO's durchzuführen. Schließlich wird die Realisiert-Beziehung zwischen ADO und seiner Vater-Systemkomponente gelöst.

Operation einer Repräsentierungs-Komponente

Beim Löschen einer Operation OP einer Repräsentierungs-Komponente ADO sind zunächst in den zur Realisierungs-Variante gehörenden Implementations-Varianten sämtliche aktuellen Zugriffe auf ADO mittels OP als zu löschend zu markieren. Abb. 6-18 identifiziert die betroffenen Implementations-Varianten. Bruder-Systemkomponenten, die Zugriffsrecht bzgl. ADO mit ZR-Attribut OP besitzen, ist dieses ZR-Attribut zu nehmen. Anschließend kann OP aus dem Operationensatz von ADO gestrichen werden.

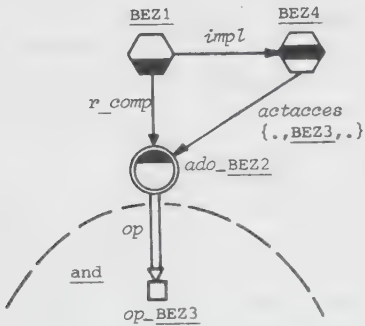


Abb. 6-18

6.6 WIEDERHERSTELLUNG DER AUSFÜHRBARKEIT

Wurden Spezifikation oder Implementation des Systems modifiziert, ist die Ausführbarkeit des Systems wiederherzustellen.

Es sei zunächst die konventionelle Systemausführung betrachtet. In konventionellen Programmiersystemen sind die nach Modifikationen neu zu übersetzenden Moduln vom Entwickler selbst zu bestimmen. Da dies zeitraubend und fehleranfällig ist, werden aus Bequemlichkeitsgründen oft das ganze System oder zumindest mehr Moduln als nötig neu übersetzt. Dabei kann der Übersetzungsaufwand beträchtlich sein. Da der vorgestellte Software-Arbeitsplatz den exakten Wirkbereich einer Modifikation kennt, kann er den notwendigen Übersetzungsaufwand generell auf ein Minimum reduzieren.

Wird in einer Implementations-Variante geändert oder gelöscht bzw. ein Änderungs- oder Lösch-Vermerk angebracht, wird der evtl. bereits existierende Bindecode zur umfassenden Modulrumpf-Variante automatisch gelöscht. Auch sämtliche Ladecodes zu System-Varianten, in die dieser Bindecode einging, werden gelöscht. Wurde die

Implementations-Variante korrigiert und enthält keine Änderungs-/Lösch-Vermerke mehr, kann der umfassende Modulrumpf neu übersetzt werden. Es sind prinzipiell zwei Übersetzungs-/Bindemodi vorgesehen. Im *kontrollierten* Modus werden Modulrumpfe nur auf ein entsprechendes Entwickler-Kommando hin übersetzt und gebunden. Sind dabei Moduln anzubinden, die noch nicht übersetzt sind, wird deren Übersetzung angestoßen. Im *automatischen* Modus dagegen werden Modulrumpfe nach Abschluß ihrer Bearbeitung sofort automatisch übersetzt¹⁾ und zum frühestmöglichen Zeitpunkt mit anderen Moduln gebunden. Auch das Nachladen kann automatisch erfolgen.

Auch nach einer Modifikation an der Systemspezifikation, die keine Modifikationen am Quelltext von Implementations-Varianten notwendig macht, kann ein neuerliches Codeerzeugen bzw. Binden erforderlich sein, das nach Abschluß der Bearbeitung der aktuellen Spezifikationseinheit automatisch angestoßen wird. Beispiele dafür seien die Eingabe oder das Löschen²⁾ einer Repräsentierungs-Komponente von einfachem Datentyp. Diese Modifikation bewirkt eine Verschiebung der Relativadressen der modullokalen Objekte der Repräsentierung. Deshalb ist für alle Modulrumpfe, die Implementations-Varianten zur modifizierten Realisierungs-Variante beinhalten, aus dem Modulrumpf automatisch neuer Bindecode zu erzeugen. Sämtliche Ladeodes zu System-Varianten, in die die ersetzten Bindecodes vorher eingingen, sind durch ein Neubinden automatisch zu ersetzen. Wird dagegen eine Repräsentierungs-Komponente eines abstrakten Datentyps *ADT* zusätzlich eingegeben oder gelöscht²⁾, ist keine Neuübersetzung, sondern nur ein Neubinden notwendig: Bezüge auf abstrakte Datenobjekte werden - wie in Abschn. 5.2 beschrieben - erst beim Binden abgesättigt. Aus diesem Grund ist auch keine Neuübersetzung erforderlich, wenn sich mit der Repräsentierung von *ADT* der Speicherbedarf der Objekte dieses Typs ändert. Das System bleibt ohne zusätzliche Maßnahmen weiterhin ausführbar, falls eine Modifikation eine echte Ergänzung bedeutet, z.B. die Hinzunahme einer neuen Operation oder eines neuen Typs.

1) Sofern dies die Regeln der Übersetzungs-Reihenfolge erlauben.

2) Die zu löschende Repräsentierungs-Komponente besitze kein angewandtes Auftreten mehr.

Für die interpretative Systemausführung sind ebenfalls obige Maßnahmen zur Wiederherstellung der Ausführbarkeit zu ergreifen, wobei jedoch jeweils das Neubinden entfällt.

Wenn bei einer zusätzlichen interpretativen Prozedurausführung ein Inkrement der Prozedurimplementationen einen Änderungs- oder Löschvermerk erhält, wird der zugehörige Inkrementcode gelöscht. Das System ist trotzdem stets ausführbar: Spätestens, wenn zur Laufzeit das Inkrement angelaufen wird, ist es zu modifizieren und zu übersetzen.

KAPITEL 7: PHASENÜBERGREIFENDE WERKZEUGE UND LITERATURVERGLEICH

In den vorangehenden Kapiteln wurde erörtert, inwieweit der Software-Arbeitsplatz IPSEN das Programmieren-im-Großen, die Systemintegration, den Systemtest und die Systempflege unterstützt. Neben dieser phasenspezifischen Unterstützung bietet IPSEN auch Werkzeuge für phasenübergreifende Aufgaben. Diese können im Rahmen dieser Arbeit nur angedeutet werden.

Im Zuge der Projektorganisation erhalten die Mitglieder eines Entwicklungsteams Zuständigkeiten (und damit Verantwortlichkeiten) angetragen: Bestimmte Entwickler sind beispielsweise ausschließlich Entwerfer, Implementatoren oder Tester. Der Zuständigkeitsbereich eines einzelnen Entwicklers wird dabei auf einzelne Systemkomponenten, einzelne Hierarchieebenen oder Teilsysteme begrenzt sein. Die Vergabe und Überwachung entsprechender Bearbeitungs-Berechtigungen wird von einem speziellen IPSEN-Werkzeug übernommen. Entsprechend seiner Bearbeitungs-Berechtigung steht einem Entwickler ein individuell zugeschnittener Kommando-Satz zur Verfügung.

IPSEN verwaltet die privaten Bibliotheken der einzelnen Entwickler sowie die Projektbibliothek und übernimmt den Austausch von Bibliothekseinträgen (Spezifikationseinheiten, Programm-Moduln, Bindecodes) zwischen diesen Bibliotheken. Beispielsweise ist ein implementierter Programm-Modul von der privaten Bibliothek des entsprechenden Implementators in die Projektbibliothek zu übertragen, falls er (nach erfolgreichem Einzeltest) zur allgemeinen Verwendung freigegeben wird. Die Freigabekontrolle wird dabei ebenfalls von IPSEN übernommen. Die Rücknahme eines Programm-Moduls aus der Projektbibliothek hat beispielsweise zu erfolgen, falls Modifikationen an der Modul-Implementation oder -Spezifikation vorzunehmen sind.

IPSEN ermöglicht es einem Entwickler weiterhin, Mitteilungen an Mit-Entwickler zu senden (z.B. Informationen über eine Schnittstellen-Erweiterung oder eine neue Systemkomponente), wobei die Empfänger gezielt ausgewählt werden (z.B. Entwickler von Systemkomponenten im Zugreifbarkeitsbereich der Systemkomponente, für die der Absender verantwortlich ist). Außerdem kann ein Entwickler einen Zettelkasten für private Notizen anlegen.

Für die Erstellung der unterschiedlichen Dokumentationen (Benutzerhandbuch, Entwicklungsdokumentation u.a.) steht ein komfortables Textsystem bereit, das auf die Beschreibung "wohlstrukturierter" Softwaresysteme zugeschnitten ist und gewisse Gliederungsschemata, Textbausteine, Merklisten usw. vorgibt. Nach jeder Modifikation der Systemspezifikation/-implementation ist die Dokumentation zu aktualisieren. IPSEN kontrolliert diese Aktualisierung und verteilt die neue Dokumentation gezielt an den entsprechenden Empfängerkreis unter den Mit-Entwicklern.

IPSEN informiert natürlich auch über den aktuellen Stand des Projekts. Die fachlichen Informationen (Stand der Systemspezifikation/-implementation bzw. des Systemtests) sind dabei den Entwicklern und den Projektmanagern entsprechend ihren Zuständigkeiten selektiv anzubieten. Managementbezogene Informationen (wie Informationen zur Projektplanung, Terminkontrolle oder Projektstatistik) stehen jedoch nur den verantwortlichen Projektmanagern zur Verfügung.

Interne Datenbasen zu den skizzierten phasenübergreifenden Leistungen von IPSEN bilden zusätzliche Graphen, wie ein *Organisationsgraph* oder ein *Dokumentationsgraph*. Diese Graphen sind mit dem Systemgraph und den Modulgraphen verflochten. Somit ergibt sich ein Implementierungskonzept, das für die phasenspezifische und phasenübergreifende Unterstützung einheitlich ist.

Im folgenden wird IPSEN kurz mit drei Vertretern bereits eingesetzter Software-Entwicklungsumgebungen verglichen.

GANDALF /HAB82/ gestattet es wie IPSEN, einzelne Systemkomponenten und ihre Quer-Beziehungen inkrementell und syntaxgesteuert zu spezifizieren, sieht jedoch keine Regeln vor, die aufgrund gewisser Entwurfsprinzipien die potentielle Vielfalt an Quer-Beziehungen einschränken. Die Modul-Implementierung erfolgt ebenfalls mit Hilfe eines syntaxgesteuerten Editors und eines inkrementellen Compilers. Diese Werkzeuge werden für verschiedene Implementierungssprachen (z.B. ADA) automatisch generiert. Die Prozeduren der Modul-Implementationen können dynamisch gebunden und geladen werden. Für den Systemtest steht ein komfortables implementierungssprach-

bezogenes Debugsystem bereit, das jedoch fürs Programmieren-im-Großen keine speziellen Funktionen anbietet. Bei Änderungen an der Export-/Import-Schnittstelle eines Moduls werden nur die Moduln im Wirkungsbereich der Änderung automatisch bestimmt, die potentiell, jedoch nicht aktuell von der Änderung betroffen sind. Komfortable Funktionen für die Versionskontrolle und für die Projektorganisation runden GANDALF ab.

Das CDL2-Labor /BBD81/ ist implementierungssprach-orientiert und unterstützt ausschließlich die Entwicklung von Softwaresystemen, die in CDL2 geschrieben werden. CDL2 bietet ein Modulkonzept an, das die oberste Ebene eines Softwaresystems als Ansammlung von beliebig vernetzten Moduln sieht. Jeder Modul ist das Ergebnis funktionaler Abstraktion und ist hierarchisch in Schichten (layers) zerlegt. Jede Schicht dieser Realisiert-Hierarchie enthält beliebig viele Abschnitte (sections), die lokalen funktionsorientierten und datenorientierten Systemkomponenten entsprechen. Zugriffe sind nur zwischen Abschnitten derselben Schicht bzw. benachbarter Schichten möglich. Die Spezifikation eines Softwaresystems ist nicht bildschirmorientiert, sondern ausschließlich zeilenorientiert erstellbar. Die Überprüfung der Konsistenz der zeilenweise eingegebenen Spezifikation eines Abschnitts erfolgt batchorientiert. Die Implementation der Abschnitte kann ebenfalls nur zeilenweise eingebracht und batchorientiert übersetzt werden. Die Systemausführung erfolgt konventionell. Nach Modifikationen an der Systemspezifikation werden nur die Systemeinheiten bestimmt, die potentiell, jedoch nicht aktuell von der Modifikation betroffen sind.

PLASMA/D /FKW81/ unterstützt die Erstellung der Spezifikation eines modularen Softwaresystems, die in einer gängigen Implementierungssprache (z.B. PASCAL) formuliert wird. Die Strukturmodelle von PLASMA/D und IPSEN ähneln sich. Die Spezifikation kann inkrementell und syntaxgesteuert eingebracht werden, wobei die verwendete Menuetechnik etwas umständlich erscheint. Die Programm-Moduln sind nur batchorientiert übersetzbar. Die Test-Werkzeuge sehen keine speziellen Funktionen fürs Programmieren-im-Großen vor. Die Wiederherstellung der Systemkonsistenz nach Modifikationen erfolgt nur unter Berücksichtigung potentieller, nicht aktueller Auswirkungen. Der Bereich der Projektorganisation erfährt durch PLASMA/D komfortable Unterstützung.

SCHLUSSBEMERKUNG

Diese Arbeit soll enden mit einem besonderen Dank an Prof. Dr. M. Nagl (Osnabrück), den Initiator des Projekts IPSEN, und an Prof. Dr. H. J. Schneider für die intensive Betreuung der Arbeit und ihre ständige Diskussionsbereitschaft.

Ich danke auch Prof. Dr. H. Müller für die Übernahme des Zweitgutachtens.

Mein Dank gilt weiter G. Engels und W. Schäfer (beide Osnabrück) für die gute Zusammenarbeit und viele fruchtbare Diskussionen sowie meinen Kollegen H. Göttler, J. Grosch, Th. Küppers, M. Messerer, K. Pelz, M. Reitenspieß und A. Völk, die durch zahlreiche Hinweise zu dieser Arbeit beigetragen haben.

Zu danken ist auch Frau E. Lührs für die mühevollen Anfertigung der Reinschrift.

STICHWORTVERZEICHNIS

- Abmagerung 27
- ADO-Deklaration
 - ~ mittels explizitem ADT 27
 - ~ mittels GADT 30
 - ~ mittels implizitem ADT 27
- ADO-Parameter 69
- ADT-äquivalent 37
- ADT-Deklaration
 - ~ mittels GADT 30
 - ~ mittels Typdefinition 27, 30
- ADT-Parameter 72
- Anwendbarkeitsbedingung
 - negative ~ 130
 - positive ~ 130
- Arbeits-Bereich
 - ~ des Bildschirms 98
- ASSMEM 19
- Attributierung
 - aktuelle ~ 83
- Attribut-Regel 128
- Attributwert
 - aktueller ~ 83
- Ausprägung 29
 - partielle ~ 30
- benutzbar 26, 41
- Benutzbarkeitsbereich 66
- Benutzung
 - aktuelle ~ 64, 75
 - potentielle ~ 64
- Benutzungsbereich 65
- Benutzungskontrolle 63
- Benutzungsrecht 64
 - explizites ~ 67
 - implizites ~ 66
 - virtuelles ~ 67
- Benutzungsrecht-Beziehung 64
- Benutzungsspezifikation 61
- bottom-up-Systementwurf 108
- BR-Objekt 64
- BR-Parameter
 - aktueller ~ 67
 - formaler ~ 67
- BR-Subjekt 64
- Codieren-im-Großen 4, 152
- Compiler
 - inkrementeller ~ 171
- Datenabstraktion 13
- Datenobjekt
 - abstraktes ~ 13, 21
 - einfaches ~ 22
- Datentyp
 - abstrakter ~ 13, 26
 - einfacher ~ 22
 - generischer abstrakter ~ 26
- Definiert-Beziehung 47
- Dokumentationsgraph 209
- EDIT-Zeiger 121
- EDO-Parameter 68
- EDT-Parameter 71
- Einbettungsüberführung 127
 - identische ~ 128
- Entwicklungsmodus
 - batchorientierter ~ 97
 - inkrementeller ~ 114
- Funktion
 - abstrakte ~ 13, 39
- Funktionsklasse 41
 - generische ~ 46
 - parametrisierte ~ 42
- GADT-Deklaration
 - ~ mittels GADT 30
 - ~ mittels Typdefinition 30
- GADT-strukturiert 37
- Graph
 - hierarchischer ~ 47
- Graph-Grammatik 10, 127
 - attributierte ~ 128
 - programmierte ~ 130
 - zweistufige ~ 129
- H-Graph 47
- Hyper-Produktion 128
- Implementations-Attribut 172
- Implementations-Variante 172
- Implementierungsmodus
 - batchorientierter ~ 167
 - inkrementeller ~ 170
 - syntaxgesteuerter ~ 170
- information hiding 13, 18
- Informations-Bereich
 - ~ des Bildschirms 98
- Informationsverteilung
 - kontrollierte ~ 18
- Initialisierung 46
- Inkrement 114, 116
 - Ändern eines ~ 119
 - aktuelles ~ 121
 - alternatives ~ 116

- atomares ~ 116
- Eingabe eines ~ 118
- Erzeugen eines ~ 118
- globales ~ 117
- komplexes ~ 116
- Löschen eines ~ 118
- lokales ~ 117
- obligates ~ 116
- optionales ~ 116
- repetitives ~ 116
- Inkrement-Kommando 122
- Inkrementoperation 118

- Kantenmarkierungs-Alphabet 127
 - terminales ~ 127
- Kantenmarkierungs-Attribut 82
- Knotenmarkierungs-Alphabet 127
 - nichtterminales ~ 127
 - terminales ~ 127
- Knotentyp-Attribut 82
- Kommando-Interpreter 132
- Konkretisierung 29
- Konsistenzbedingungen 83
- Konsistenzprüfungs-Reihenfolge 103
- Kontrollprozedur 130

- Listenorganisator 113

- Maschine
 - abstrakte ~ 15
- Meta-Produktion 128
- Modularisierung
 - logische ~ 12
 - physische ~ 14
- Modulgraph 10, 167
- Modulkonzept 152
- Modulrumpf 14, 152
- Modulschema 153

- Objekt-Parameter 68
- Organisationsgraph 10, 209

- Parameter
 - generischer ~ 26, 29
- Platzhalter 120
 - aktueller ~ 121
- *-Platzhalter ~ 120
- Problemanalyse 3
- Problembeschreibung 3
- Produktion 127
- Programm-Editor
 - syntaxgesteuerter ~ 172
- Programm-Entwicklungsumgebung 6
 - Programm-Erstellungswerkzeuge 7
 - Programmieren-im-Großen 5
 - Programmieren-im-Kleinen 5
 - Programm-Modul 14, 152
 - Projektion 81
 - Prozedurausführung
 - interpretative ~ 171
 - quasi-H-Graph 58
 - Realisiert-Beziehung 17, 47
 - Realisierung 12
 - Realisierungs-Attribut 79
 - Realisierungs-Variante 77
 - Repräsentierung 13
 - Ressource 12
 - \$_Pfad 81
 - Schnittstelle
 - äußere ~ 12
 - Export-/Import-~ 14, 152
 - innere ~ 12
 - virtuelle äußere ~ 28
 - virtuelle innere ~ 29
 - separate-Klausel 100
 - SE-Stummel 100
 - SK-Spezifikation 100
 - Software-Arbeitsplatz 6, 7
 - Software-Engineering
 - Prinzipien des ~ 11
 - Software-Entwicklungsumgebung 6
 - Software-Entwicklungszyklus 6
 - Software-Erstellungswerkzeuge 6
 - Softwaresystem
 - datenorientiertes ~ 51
 - funktionsorientiertes ~ 51
 - Spezifikations-Checker 98
 - inkrementeller ~ 115
 - Spezifikations-Editor
 - syntaxgesteuerter ~ 115
 - Spezifikationseinheit 99
 - aktuelle ~ 109
 - Spezifizierbarkeit
 - getrennte ~ 99
 - Streckung 48
 - strukturähnlich 37
 - Strukturmodell 18
 - System-Attribut 173
 - Systemausführung
 - interpretative ~ 175
 - konventionelle ~ 173
 - Systementwurf 3
 - Systemgraph 10, 82
 - bzgl. aktueller Benutzungen konsisten-

- ter ~ 96
 - bzgl. der expl. Benutzungsrecht-
 - Beziehung konsistenter ~ 94
 - bzgl. der expl. Zugriffsrecht-
 - Beziehung konsistenter ~ 88
 - hierarchischer ~ 85
 - ZR-konsistenter ~ 94
 - Systemgraph-Test 126
 - Systemgraph-Transformation 126
 - Systemhierarchie 16, 49
 - Systemimplementation 5
 - Systemintegration 5
 - Systemkomponente 12, 20
 - datenorientierte ~ 12
 - funktionsorientierte ~ 12
 - Systempflege 5
 - Systemspezifikation 11
 - semantische ~ 4
 - syntaktische ~ 4
 - Systemstruktur
 - Semantik der ~ 3
 - Syntax der ~ 3
 - Systemstruktur-Interpretierer 175
 - Systemtest 5
 - System-Variante 173
 - Teilhierarchie 48
 - Teilsystem 17, 48
 - durch eine Systemkomponente aufgespanntes ~ 77
 - Teilsystem-Variante 77
 - durch eine Realisierungs-Variante aufgespannte ~ 78
 - Testprozedur 130
 - top-down-Systementwurf 108
 - typäquivalent 36
 - Typisierung
 - strenge ~ 36
 - Typ-Parameter 68
 - Übersetzungs-/Bindemodus
 - automatischer ~ 206
 - kontrollierter ~ 206
 - Unparse-Verfahren 98
 - Vater-Einheit 100
 - Waldstruktur 81
 - Werkzeuge 6
 - Wirkbereich
 - ~ einer Modifikation 179
 - Zeiger
 - interner ~ 136
 - Zeiger-Kommando 121
 - Zeiger-Produktion 136
 - ZR-Attribut 53
 - ZR-Objekt 53
 - ZR-Parameter
 - aktueller ~ 61
 - formaler ~ 61
 - ZR-Subjekt 53
 - zugreifbar 21
 - Zugreifbarkeitsbereich 55
 - Zugriff
 - aktueller ~ 54
 - faktischer ~ 54
 - potentieller ~ 53
 - Zugriffsbereich 55
 - Zugriffskontrolle 18
 - Zugriffsmodell 51
 - Zugriffsoperation 12
 - Zugriffsrecht 53
 - explizites ~ 56
 - implizites ~ 56
 - Vererbung eines ~ 55
 - virtuelles ~ 56
- Zugriffsrecht-Beziehung 53
 - explizite ~ 85
 - zugriffssicher 51
 - Zugriffstransparenz 18
 - zyklenfrei 81

LITERATURVERZEICHNIS

- /ALT78/ ALTMANN, W.: "Beschreibung von Programmmoduln zum Entwurf zuverlässiger Software", Dissertation, Arbeitsber. d. Inst. f. Math. Maschinen u. DV, Univ. Erlangen-Nürnberg, Band 11, Nr. 16, 1978
- /BACR72/ BAYER, R./MC CREIGHT, E.M.: "Organization and Maintenance of Large Ordered Indexes", Acta Informatica, Vol. 1, S. 173 - 189, 1972
- /BBD81/ BAYER, M./BÖHRINGER, B./DEHOTTAY, J.P./FEUERHAHN, H./JASPER, J./KOSTER, C.H.A./SCHMIEDECKE, U.: "Software Development in the CDL2 Laboratory", in: Hünke, M.: "Software Engineering Environments", North-Holland Publishing Company, 1981
- /BRE77/ BRENDDEL, W.: "Maschinencode-Erzeugung bei inkrementeller Compilation", in: Arbeitsber. d. Inst. f. Math. Maschinen u. DV, Univ. Erlangen-Nürnberg, Band 10, Nr. 8, 1977
- /DDH72/ DAHL, O.J./DIJKSTRA, E.W./HOARE, C.A.R.: "Structured Programming", Academic Press, London, 1972
- /DEKR76/ DeREMÉR, F./KRON, H.H.: "Programming-in-the-Large versus Programming-in-the-Small", in: IEEE Transactions on Software Engineering 2, 2, S. 80 - 86, 1976
- /DoD80/ Department of Defense: "Reference Manual for the ADA Programming Language (Proposed Standard Document)", United States Department of Defense, 1980
- /ENSC82/ ENGELS, G./SCHÄFER, W.: "Specification of a Programming Support Environment by Graph Grammars", in: "Proceedings of the 8th Conference on Graphtheoretic Concepts in Computer Science (WG' 82)", Hanser Verlag, München 1982
- /ENW74/ ENCARNACAO, J./WECK, G.: "Eine Implementierung von DATAS", Institut für Angewandte Mathematik und Informatik, Univ. des Saarlandes, Bericht Nr. A 74-1, 1974
- /FKW81/ FUCHS, W./KURZ, E./WEBER, D.: "PLASMA/D: Rechnergestützte Systementwicklung auf der TA1630", Triumph-Adler AG, 1981
- /GAL78/ GALL, R.: "Umstrukturierung des softwaremäßig implementierten Assoziativspeichers DATAS auf eine 32-Bit-Maschine", Diplomarbeit am Lehrstuhl f. Programmiersprachen und Compiler, Univ. Erlangen-Nürnberg, 1978
- /GAL82/ GALL, R.: "Structured Development of Modular Software Systems: The Module Graph as Central Data Structure", in: Proceedings of the 7th Conference on Graphtheoretic Concepts in Computer Science (WG' 81)", Hanser Verlag, München, 1982
- /GAN81/ GALL, R./NAGL, M.: "Software-Implementation assoziativer Speicher", in: Elektronische Rechenanlagen 23, Heft 2, S. 61 - 71, 1981
- /GÖT77/ GÖTTLER, H.: "Zweistufige Graphmanipulationssysteme für die Semantik von Programmiersprachen", Dissertation, Arbeitsber. d. Inst. f. Math. Maschinen u. DV, Univ. Erlangen-Nürnberg, Band 10, Nr. 12, 1977

- /GÖT82/ GÖTTLER, H.: "Attributed Graph Grammars for Graphics", erscheint in: Proceedings of the 2nd Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, Springer Verlag, Berlin
- /GUT77/ GUTTAG, J.: "Abstract Data Types and the Development of Data Structures", Communic. of the ACM, 20, Nr. 6, S. 396 - 404, 1977
- /HAB82/ HABERMANN, N. u.a.: "The Second Compendium of GANDALF Documentation", Department of Computer Science, Carnegie-Mellon Univ., Pittsburgh, 1982
- /LRS79/ LEVITT, K.N./ROBINSON, L./SILVERBERG, B.A.: "The HDM Handbook", Vols. 1-3, Computer Science Lab., SRI International, Menlo Park, California, 1979
- /LIZI74/ LISKOV, B./ZILLES, S.: "Programming with Abstract Data Types", in: SIGPLAN Notices, 9, Nr. 4, S. 50 - 59, 1974
- /NAG79/ NAGL, M.: "Graph-Grammatiken: Theorie, Anwendungen, Implementierung", Vieweg-Verlag, Braunschweig, 1979
- /NAG80/ NAGL, M.: "An Incremental Compiler as Component of a System for Software Generation", in: Informatik-Fachberichte 25, Springer Verlag, Berlin, S. 29 - 44, 1980
- /NAG82/ NAGL, M.: "Einführung in die Programmiersprache ADA", Vieweg Verlag, Braunschweig, 1982
- /PAR71/ PARNAS, D.L.: "Information Distribution Aspects of Design Methodology", in: Proceedings IFIP Congress, Ljubljana, S. 339 - 344, 1971
- /PAS75/ PARNAS, D.L./SIEWIOREK, D.P.: "Use of Concept of Transparency in the Design of Hierarchically Structured Systems", in: Communic. of the ACM, 18, Nr. 7, S. 401 - 408, 1975
- /PRA69/ PRATT, T.W.: "A Hierarchical Graph Model of the Semantics of Programs", Proceed. AFIPS, Summer Joint Comp. Conference, 1969
- /ROSC77/ ROSS, D.T./SCHOMANN, K.E.: "Structured Analysis of Requirements Definition", in: IEEE Transactions on Software Engineering 3, 1, S. 6 - 15, 1977
- /SCH75a/ SCHNEIDER, H.J.: "Compiler - Aufbau und Wirkungsweise", de Gruyter Verlag, Berlin, New York, 1975
- /SCH75b/ SCHNEIDER, H.J.: "Syntax-directed Description of Incremental Compilers", Lecture Notes in Computer Science 26, Springer Verlag, Berlin, S. 192 - 201, 1975
- /STA76/ STAY, J.F.: "HIPO and Integrated Program Design", IBM Systems Journal, 15, Nr. 2, S. 143 - 154, 1976
- /WES82/ WESTFECHTEL, B.: "Operationen zur inkrementellen Modifikation des Systemgraphen", Studienarbeit am Lehrstuhl f. Programmiersprachen und Compiler, Univ. Erlangen-Nürnberg, 1982
- /WIR80/ WIRTH, N.: "MODULA-2", Technischer Report 36, Inst. f. Informatik, ETH Zürich, 1980
- /WULO76/ WULF, W.A./LONDON, R.L./SHAW, M.: "An Introduction to the Construction and Verification of ALPHARD Programs", in: IEEE Transactions on Software Engineering 2, 4, S. 253 - 265, 1976

SYSTEM-SPEZIFIKATION

```

system_spez ::=
    ado_dekl_m_impl_adt |
    adt_dekl_m_typdef |
    gadt_dekl_m_typdef |
    fk_dekl |
    pfk_dekl

```

SK-SPEZIFIKATION

```

(S.1)  ado_dekl_m_impl_adt ::=
        [sk_sep_klaus]
        ado ado_bez1)
        ado_aeuss_schnittst
        ado_realisvarianten
        end ado ado_bez2)

```

```

(S.2)  adt_dekl_m_typdef ::=
        [sk_sep_klaus]
        adt adt_bez
        adt_aeuss_schnittst
        adt_realisvarianten
        end adt adt_bez

```

```

(S.3)  gadt_dekl_m_typdef ::=
        [sk_sep_klaus]
        gadt gadt_bez ( gadt_f_par_dekl {; gadt_f_par_dekl} )
        gadt_aeuss_schnittst
        gadt_realisvarianten
        end gadt gadt_bez

```

1) Der weitere syntaktische Aufbau von Bezeichnern und Namen bleibt hier unspezifiziert.

2) Muß mit dem Bezeichner hinter ado übereinstimmen.

```

(S.3.2)  gad_t_f_par_dekl    ::=
          f_adopar_dekl  |
          f_adtpar_dekl  |
          f_edopar_dekl  |
          f_edtpar_dekl

(S.3.3)  f_adopar_dekl    ::=
          ado ado_bez {, ado_bez} : adt_name
                                     by [op_bez {, op_bez}]

(S.3.4)  f_adtpar_dekl    ::=
          adt adt_bez {, adt_bez} [: gad_t_name
                                     by [op_bez {, op_bez}]]

(S.3.5)  f_edopar_dekl    ::=
          edo edo_bez {, edo_bez} : edt_name

(S.3.6)  f_edtpar_dekl    ::=
          edt edt_bez {, edt_bez}

(S.4)    sk_sep_klaus      ::=
          separate SKrealis_name ;

(S.5)    ADO(G)ADT-aeuss_schnittst ::=
          external interface
          ADO(G)ADT_operationen
          [potzugr_potben]
          end external interface

(S.6)    ADO(G)ADT_operationen ::=
          provides
          op_spez
          {op_spez}

(S.6.1)  op_spez          ::=
          op op_bez ([op_f_par_dekl {, op_f_par_dekl}]) ;

```

```

op_f_par_dekl    ::=
    f_edopar_dekl |
    f_adopar_dekl

```

```

(S.7) potzuger_potben    ::=
    access/use rights
    {potzuger_spez}
    {(g)adt_potben_spez}
    {(p)fk_potben_spez}

```

```

(S.7.1) potzuger_spez    ::=
    ado ado_name {, ado_name1} by [op_bez {, op_bez}];

```

```

(S.7.2) (g)adt_potben_spez    ::=
    (g)adt (g)adt_name2 {, (g)adt_name}
    by [op_bez {, op_bez}];

```

```

(S.7.3) (p)fk_potben_spez    ::=
    (p)fk (p)fk_name3 {, (p)fk_name}
    by [fu_bez {, fu_bez}];

```

```

(S.8) SK_realisvarianten    ::=
    realizations
    SK_realisvariante
    {SK_realisvariante}

```

```

(S.8.1) SK_realisvariante    ::=
    SK_realis_rumpf |
    SK_realis_verwlist

```

```

(S.8.2) ado_realis_verwlist    ::=
    ado-realization adorealiz_bez {, adorealiz_bez}
    : separate ;

```

analog für adt, gadt.

¹) Sämtliche abstrakten Datenobjekte der Liste müssen typäquivalent sein.

²) Abkürzung für die Alternative adt_name/gadt_name.

³) Abkürzung für die Alternative fk_name/pfk_name.

```

(S.9)  fk_dekl    ::=
        [sk_sep_klaus]
        fk fk_bez
            fk_aeuss_schnittst
            fk_realisvarianten
        end fk fk_bez

(S.10)  pfk_dekl  ::=
        [sk_sep_klaus]
        pfk pfk_bez ( pfk_adopar_repr {; pfk_adopar_repr} )
        pfk_aeuss_schnittst
        pfk_realisvarianten
        end pfk pfk_bez

        pfk_adopar_repr  ::=
            set of ado_bez : (g)adt_name

(S.11)  (P)FK_aeuss_schnittst  ::=
        external interface
            (P)FK_funktionen
            potzugr_potben
        end external interface

(S.12)  (P)FK_funktionen  ::=
        provides
            funk_spez
            {funk_spez}

(S.12.1)  funk_spez  ::=
            func fu_bez ( [funk_f_par_dekl {; funk_f_par_dekl}] );

            funk_f_par_dekl  ::=
                f_edopar_dekl |
                f_adopar_dekl1) |
                f_aehnl_adopar_dekl1)

```

¹⁾ Nur zulässig, falls die Funktion einer PFK angehört.

```
f_aehn1_adopar_dekl    ::=
    ado ado_bez {, ado_bez} : gadt_name
                                by [op_bez {, op_bez}]
```

SK-REALISIERUNGS-VARIANTE

```
(RV.1)  ado_realis_rumpf    ::=
    [ado_skrealis_sep_klaus]
    ado-realization adorealiz_bez
                    ado_inn_schnittst
    end ado-realization adorealiz_bez1)
```

(analog für adt, gadt, fk, pfk)

```
(RV.2)  SK_skrealis_sep_klaus    ::=
    separate SK_name ;
```

```
(RV.3)  ADO(G)ADT_inn_schnittst    ::=
    internal interface
        repraesentierung
        lok_typen
        lok_funktionen
        ben_spezifikationen
    end internal interface
```

```
(RV.4)  (P)FK_inn_schnittst    ::=
    internal interface
        lok_typen
        lok_funktionen
        ben_spezifikationen
    end internal interface
```

¹⁾ Muß mit dem Bezeichner hinter ado-realization übereinstimmen.

(RV.5) repraesentierung ::=

representation

repr_komp_dekl

{repr_komp_dekl}

(RV.5.1) repr_komp_dekl ::=

ado_dekl_m_impl_adt |

ado_dekl_m_impl_adt_verwlist |

ado_dekl_m_expl_adt |

ado_dekl_m_gadt |

edo_dekl |

dyn_repr

(RV.5.2) ado_dekl_m_impl_adt_verwlist ::=

ado ado_bez {, ado_bez} : separate ;

(RV.5.3) ado_dekl_m_expl_adt ::=

ado ado_bez {, ado_bez} : adt_name

provides op_bez {, op_bez};

(RV.5.4) ado_dekl_m_gadt ::=

ado ado_bez {, ado_bez} :

gadt_name (gadt_a_par_list {; gadt_a_par_list})

provides op_bez {, op_bez} ;

(RV.5.4.1) gadt_a_par_list ::=

a_adopar_list |

a_adtpar_list |

a_edopar_list |

a_edtpar_list

(RV.5.4.2) a_adopar_list ::=

ado_name {, ado_name}¹⁾

¹⁾ Aktuelle Parameter müssen mit den entspr. formalen Parametern typäquivalent sein.

```
(RV.5.4.3)  a_adtpar_list  ::=
              adt_name {, ado_name}1)

(RV.5.4.4)  a_edopar_list  ::=
              edo_name {, edo_name}2)

(RV.5.4.5)  a_edtpar_list  ::=
              edt_name {, edt_name}

(RV.5.5)    edo_dekl      ::=
              edo edo_bez {, edo_bez} : edt_name ;

(RV.5.6)    dyn_repr      ::=
              dynamic

(RV.6)      lok_typen     ::=
              local data types
              {lok_typ_dekl}

(RV.6.1)    lok_typ_dekl  ::=
              (g)adt_dekl_m_gadt |
              (g)adt_m_typdef_verwlist |
              edt_dekl

(RV.6.2)    (g)adt_dekl_m_gadt  ::=
              (g)adt (g)adt_bez {, (g)adt_bez}
              [( gadt_f_par_dekl {; gadt_f_par_dekl} )]3) :
              gadt_name ( gadt_a_par_list {; gadt_a_par_list} )
              provides op_bez {, op_bez} ;

(RV.6.3)    edt_dekl      ::=
              edt edt_bez {, edt_bez} ;
```

¹⁾ Aktuelle Parameter müssen zu den entspr. formalen Parametern strukturähnlich sein.

²⁾ Aktuelle Parameter müssen vom selben EDT sein wie die formalen Parameter.

³⁾ Im Falle eines GADT's notwendig.

```

(RV.7) lok_funktionen ::=
    local functions
    { (p)fk_dekl_verwlist }
    { funk_spez }

(RV.7.1) (p)fk_dekl_verwlist ::=
    (p)fk (p)fk_bez {, (p)fk_bez} : separate ;

(RV.7.2) ben_spezifikationen ::=
    use-specifications
    { ben_spez }

(RV.7.3) ben_spez ::=
    pfk_name ( potzugr_spez
              { potzugr_spez } ) ;

```

META-PRODUKTIONEN

```

(M1)  SK ::=  ado | adt | gadt | fk | pfk
(M2)  ADO(G)ADT ::=  ado | (G)ADT
(M3)  (G)ADT ::=  adt | gadt
(M4)  (P)FK ::=  fk | pfk

```

Lebenslauf (tabellarisch)

Rupert Gall

Wohnsitz: 8520 Erlangen, Kulmbacher Str.6

Geboren als erstes von 3 Kindern des 1972 verstorbenen Landgerichtsleiters Rudolf Gall und seiner Frau Pia Gall, geb. Zaengerle, am 12. Dezember 1952 in Regensburg.

Bekenntnis: Römisch-katholisch

Volksschule: September 1959 - Juli 1961 : Schule Hl. Kreuz,
Regensburg
September 1961 - Juli 1963 : Schule West,
Regensburg

Gymnasium: September 1963 - Juli 1972 :
Albrecht-Altdorfer-Gymnasium, Regensburg
Erhalt des Reifezeugnis am 23.6.1972

Universität: Friedrich-Alexander-Universität Erlangen-Nürnberg
Aufnahme des Studiums der Informatik (Fachrichtung
Diplom) im Wintersemester 1972/73 und des Studiums
der Mathematik (Fachrichtung Diplom) im Winterse-
mester 1973/74

12.11.1974 : Diplom-Vorprüfung Informatik
25.11.1975 : Diplom-Vorprüfung Mathematik
11.10.1978 : Abschluss des Studiums der Informatik
(Erhalt des Diplom-Zeugnisses)

Beschäftigt seit 1.3.1979 als wissensch. Hilfskraft am Lehr-
stuhl für Programmiersprachen und Compiler, Prof.
Dr. H. J. Schneider, Erlangen, Martensstr. 3

In letzter Zeit sind folgende Arbeitsberichte des Instituts für
Mathematische Maschinen und Datenverarbeitung erschienen:
=====

Band 13

=====

- Nr. 1 Schrammel, D., Bayer, W., Gliemann, H.-P., Heigl, M.,
Mertens, P.:
Ein Leistungs- und Wirtschaftlichkeitsvergleich zwischen
Klein- und Großcomputern
(Januar 1980)
- Nr. 2 Göttler, H.:
Semantische Modelle
(März 1980)
- Nr. 3 Fromm, H.J.:
Zur Modellierung der Speicherinterferenz bei hierarchisch
organisierten Multiprozessorsystemen
(April 1980)
- Nr. 4 Jäpel, D.:
Klassifikatorbezogene Merkmalsauswahl
(Juli 1980)
- Nr. 5 Bodendorf, F., Mertens, P., Haas, R., Weber, G. Wolf, G.:
SAMBA - ein Methodenbankrahmen um das Statistik-Paket SPSS
(Juli 1980)
- Nr. 6 Bartsch, B.:
Inferenz und Analyse spezieller Graphgrammatiken für die
syntaktische Mustererkennung
(September 1980)
- Nr. 7 Wurm, F.X.:
Auftragssystem für eine Mehrprozessoranlage
(Oktober 1980)
- Nr. 8 Schedelbeck, A.:
Vergleich der beiden höheren Realzeitprogrammiersprachen
ADA und PEARL.
(Oktober 1980)
- Nr. 9 Jahresbericht 1980 der Informatik.
(März 1981)

- Nr. 1 Pattern Recognition
(Research at Lehrstuhl für Informatik 5)
(Februar 1981)
- Nr. 2 Bunke, H.:
Analyse elektrischer Schaltpläne
(März 1981)
- Nr. 3 Linster, C.-U.:
SYMPOS/UNIX - Ein Betriebssystem für homogene
Polyprozessorsysteme
(Juni 1981)
- Nr. 4 Akyildiz, J.F., Bolch, G.:
Analytic Solution Techniques for Queueing Network
Models of Computer Systems
(Juli 1981)
- Nr. 5 Lindstedt, W.G.:
Ein Verfahren zur Erstellung portabler algorithmischer
Laufzeitprogramme
(Juli 1981)
- Nr. 6 Vollmar, R.:
Zum Begriff des Parallelismus bei Polyautomaten
(Juli 1981)
- Nr. 7 Luft, A.L.:
Rationaler Sprachgebrauch und orthosprachliche
Standardisierung als Grundlagen zuverlässiger Software-
Entwicklung und -Dokumentation
(Januar 1982)

Band 14 Fortsetzung
=====

Nr. 8 Händler, W.:

PROCEEDINGS OF A WORKSHOP ON TAXONOMY IN COMPUTER
ARCHITECTURE

(Februar 1982)

Nr. 9 Beth, Th., Hess, P., Wirl, K.:

Materialien zur Kryptographie

(März 1982)

Nr. 10 Jahresbericht 1981 der Informatik

(März 1982)

- Nr. 1 Wittmann, A.:
Ein Mechanismus für die Synchronisation paralleler
Prozesse
(Februar 1982)
- Nr. 2 Maehle, E.:
Fehlertolerantes Verhalten in Multiprozessoren -
Untersuchungen zur Diagnose und Rekonfiguration
(März 1982)
- Nr. 3 Bathelt, P.:
Vergleich von Synchronisationsmechanismen
(Mai 1982)
- Nr. 4 Keramidis, S.:
Eine Methode zur Spezifikation und korrekten
Implementierung von asynchronen Systemen
(Juni 1982)
- Nr. 5 Fromm, H.J.:
Multiprozessor-Rechenanlagen:
Programmstrukturen, Maschinenstrukturen und Zuordnungs-
probleme
(Mai 1982)
- Nr. 6 Bley, H.:
Vorverarbeitung und Segmentierung von Stromlaufplänen
unter Verwendung von Bildgraphen
(Juli 1982)
- Nr. 7 Hohl, W.:
Informatik-Sammlung
Katalog mit historischen Erläuterungen
(August 1982)
- Nr. 8 Messerer, M.:
Ein neuer Ansatz zur Parallelisierung von
Compilern
(August 1982)

- Nr. 9 Grosch, J.:
Eine Programmiersprache mit mengentheoretischen
Konstrukten und deren effiziente Implementierung
(August 1982)
- Nr.10 Kleinöder, W.:
Stochastische Bewertung von Aufgabenstrukturen
für hierarchische Mehrrechnersysteme
(August 1982)
- Nr.11 Kneißl, F.:
Realisierung von Makro-Datenflußmechanismen auf
hierarchischen Mehrrechnersystemen
(August 1982)
- Nr.12 Brendel,W.,Fan,Z.,Klar,R.,Schmielau,W.:
ERES 82
Handbuch und Fallstudie
Handbook and Case Study
(Oktober 1982)
- Nr.13 Pelz, K.:
Portable Codegenerierung in Übersetzern
für Prozess-Programmiersprachen
(Oktober 1982)
- Nr.14 Sommer,U.,Steinchen,J.:
Ein Kleinrechner-Grafiksystem zur Dokumentation
von EDV-Abläufen
(Oktober 1982)
- Nr.15 Hein, H.-W.:
Das Erlanger Spracherkennungssystem
(November 1982)

- Nr. 16 Bolch, G., Bruchner, W.:
 Analytische Modelle für symmetrische Mehrprozessor-
 anlagen mit dynamischen Prioritäten
 (Dezember 1982)
- Nr. 17 Jahresbericht 1982 der Informatik
 (März 1983)

